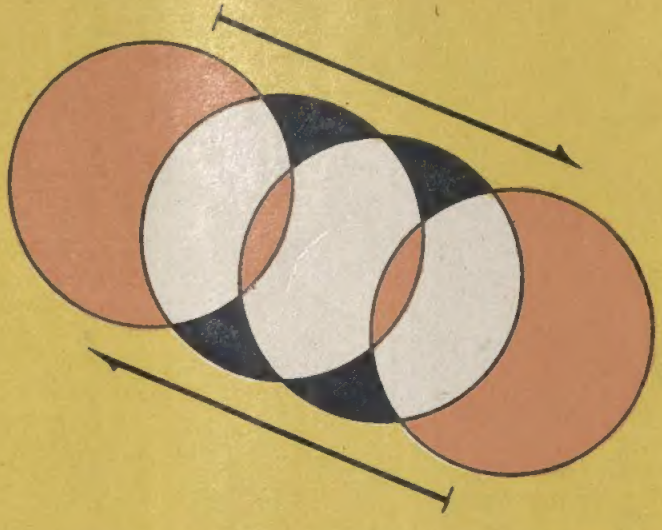
# DIGITAL COMPUTER DESIGN PRINCIPLES

*With*
*Introduction to Microprocessors*

**M. R. BHUJADE**



**PITAMBAR PUBLISHING COMPANY**

# DIGITAL COMPUTER DESIGN PRINCIPLES

## With Introduction to Microprocessors

## ABOUT THE BOOK

This book systematically develops design concepts from elementary digital components to the complex systems like processors and memories. The book, besides serving as a text on Digital Computer Hardware can also serve the electronic/computer engineers in industry, who have to deal with computer hardware or computer (or microprocessor) based equipments. The highlights of the book are : Provides comprehensive treatment on computer arithmetic; gives indepth treatment to semi-conductor memories; covers various addressing techniques and other architectural aspects; illustrates the CPU design principles by giving complete design of a CPU; discusses in detail the I/O subsystems and error detection and correction techniques; gives microcomputer design using popular 8080/8085 microprocessors; has a large number of exercises many of which could be used as laboratory projects and experimental set-ups.

## ABOUT THE AUTHOR

DR.M.R. BHUJADE served as a Computer Engineer in the Tata Institute of Fundamental Research, Bombay, for a brief period and later joined Indian Institute of Technology, Bombay, where presently he is Assistant Professor in Computer Science.

# DIGITAL COMPUTER DESIGN PRINCIPLES
## With Introduction to Microprocessors

**M. R. BHUJADE**
*Ph.D.*
*Assistant Professor*
*Department of Computer Science and Engineering,*
*Indian Institute of Technology, Bombay*

**PITAMBAR PUBLISHING COMPANY**
*EDUCATIONAL PUBLISHERS*
**888, EAST PARK ROAD, KAROL BAGH,**
**NEW DELHI - 110005 (INDIA)**

Published by
**PITAMBAR PUBLISHING COMPANY**
Educational Publishers
888, East Park Road, Karol Bagh,
NEW DELHI – 110005 (INDIA)
Telephones:
Office : 526933, 776058, 770067
Res. : 586788, 5721321, 5715182

First Edition : 1984

Second Revised Edition : 1986

Reprinted : 1989

Price: Rs.35/-



Code No. 27047



Printed at:
**Piyush Printers Publishers Pvt. Ltd.**
G -12, Udyog Nagar,
Rohtak Road Industrial Area,
New Delhi - 110041.
Telephone: 5472440

# Preface to the Second Revised Edition

The first edition of this book has been used by many colleges and universities. From the comments I received from various teacher friends and students, I am happy that the book has served a dual role, first being that of basic principles on digital computer hardware design while the second of microprocessor. Although the chapter on microprocessors was not intended initially but it was included due to importance of this new technology. Keeping these in view a complete new chapter has been included in the second edition which deals with microprocessor I/O interfacing. The inclusion of this chapter will help many in understanding and building microprocessor based systems, which forms a technology of today sweeping the entire world. I thank my publishers for bringing out the new edition.

M.R. Bhujade

# Preface to the First Edition

This book has evolved through the author's experience in teaching the undergraduate nd postgraduate courses in Digital Computers, Digital Computer Components, Digital omputer Architecture and Digital Computer Fundamentals at Indian Institute of echnology, Bombay. The material presented in this book is class room tested through a eries of lecture notes.

The chapters on Number Systems, Boolean Algebra, Logic Design and Digital Systems uilding Blocks form the pre-requisite to understand the Digital Computer Design Principles nd as such must be mastered thoroughly. There are number of chapters on computer rithmetic. The purpose of devoting separate chapters on arithmetic is three-fold. Firstly, he depth in which the topics on arithmetic are discussed is higher than that one would find in ny text book on the subject. Secondly, with the advent of microprocessor, the detailed nowledge of these topics is required by a microprocessor based system designer, who has to evelop software programs to carry out arithmetic Thirdly, to carry out the microprorammed design of a CPU, a good knowledge of these topics is a must (guess why ?).

The chapters on memories bring out the understanding and design principles of raditional core memories as well as the latest semiconductor memories. The chapters on Computer Architecture, Processor Design and I/O Systems bring out the basic principles nvolved in working and design of digital computers with various facilities. A reader having he knowledge of the material in these chapters shall be able to understand the literature and se of any computer (including microprocessors) in the machine/assembly language level. A separate chapter on microprocessors gives complete discussion of 8080 and 8085 microrocessors. The microprocessors are chosen for discussion due to their popularity.

The book besides serving as a text on Digital Computer Hardware Design can also erve the electronic/computer engineers in industries who have to deal with computer hardvare or computer based equipments.

An undergraduate one semester course could be planned around this book covering hapters 1, 2, 3, 4, 5, 9 and introducing the material from remaining chapters briefly. A postraduate course on computer hardware design principles could be planned which may nclude all the chapters of the book. If the course assumes the background in logic circuits, hapters 3 and 4 could be omitted and more emphasis given on microprocessors.

I thank Mr. S.M. Joglekar, Dr. S.P. Navathe, Dr. S.V. Kanetkar and Dr. S. Biswas vho have taken pains to go through the various parts of the manuscript of this book.

I appreciate and acknowledge the financial support given by the Curriculum Developnent Program, I.I.T. Bombay, without which it would have been difficult to bring out this ook. I thank Co-ordinater, Curriculum Development Program, I.I.T. Bombay for his mmediate response to all my requests.

I thank Prof. J.R Issac, Dean of Students' Affairs, I.I.T. Bombay, who has alw given boost to my thoughts and ideas. I am grateful to Prof. H B. Kekre, In-Charge, Compu Centre, I.I.T. Bombay for his encouragement in this kind of work.

It was Dr. V B. Aggarwal, Professor and Head Dept. of Computer Science, D University, who introduced the idea of editing my lecture notes to make a text. I thankful to him for the same.

It would not have been possible to publish the material in chapter 14 and 15 with the permission of Intel Corporation. I express my appreciation and acknowledge the h given by Intel Corporation, U.S.A in this regard.

No words can express my gratitude to my wife Mrunalini and little son Raju, who h to miss my company on many evenings during the course of this work.

My thanks are also due to Mr. R.B. Kamble for typing the manuscript of this book.

I am thankful to my publishers for bringing out this publication in a short span time.

Suggestions for improvement of the book are welcome from readers.

M.R. Bhuj

# Contents

# 1

# INTRODUCTION

## . A DIGITAL COMPUTER

)IGITAL computer is a machine which processes digital information. Information ocessing is any useful transformation of ta to produce the output data. Early mputers (1945) were designed and used by ientists and engineers as computational ols for solving complex mathematical and merical problems. However, ability of ese machines to solve other problems was cognised later and computers were soon ed in diverse applications like banking, ventory control, pay roll, air line reserva- n systems, process control, message switch- g systems, etc.

The basic power of a digital computer lies its capability to store, retrieve and anipulate data at a very high speed. ithout committing a single mistake, com- ters can retrieve, manipulate and store illions of numbers in few seconds.

Modern digital computer system comprises number of hardware and software compo- nts. In early computers, electronic rdware of computers was built using cuum tube devices. These computers were e first generation machines. With the scovery of the transistor (a semiconductor vice), computer hardware was designed and uilt using these devices. This was the second eneration of computers. In early 60's, tegrated circuits were developed which llowed the packing of a large number of ansistors, diodes, resistors etc., in a small ntegrated Circuit (IC) chip. The third generation of computers use integrated circuits as building blocks. The latest computers (fourth generation) are built using very large scale integrated (VLSI) circuit chips, each chip having a large number of circuit components. These changes in the technology made the hardware of computers progressively cheaper which resulted in the development of computer systems with considerably higher complexities. Although the computer generations, are attributed to the hardware technology, there are implicit changes in the architectural and other features of computers through the generations.

The hardware of a computer system is composed of a number of equipments made of electronic, electromagnetic and electromechanical components. On the other hand software components of a system are 'programs' which make possible the effective utilisation of the computer system by users who are not required to have the knowledge of the electronics or inner elements of the computer system.

## 1.2. HARDWARE COMPONENTS

A digital computer is typically equipped with the following hardware components :

- *Central Processing Unit (CPU) or Processor
- *Main Memory or Primary Memory
- *Secondary Memory
- *Input and Output Devices
- *Channels

A brief introduction to the above components is presented in this section.

### 1.2.1. Central Processing Unit (CPU)

The CPU facilitates processing of the information. It executes a series of basic commands called instructions. A set of such instructions available on a processor constitutes its machine language. Each instruction in the machine language is a coded description of an operation which is to be performed by the CPU. These basic operations are seldom more complex than multiplication of two numbers. A complex operation has to be broken down into a sequence of machine language instructions so that the overall effect of these, if executed in proper sequence would yield the result. The list of instructions thus formed is called a program.

A CPU has two basic components, *viz.*, (*i*) Arithmetic Logic Unit (ALU) and (*ii*) Control Unit (CU). ALU carries out the arithmetical and logical operations on the data supplied to it by the control unit. Control unit is designed such that the processor automatically executes m language instructions stored in the memory of the computer. This i bringing the instruction from the memory into the CPU, decoding i instructing other units in the comput electronic signals to carry out their tas specified sequence depending on the i tion being executed.

### 1.2.2. Main (Primary) Memory (MM

Main memory of a computer pr storage for data and instructions on the CPU operates. For more than decades, magnetic cores have dominate storage medium used in main mem However, with the advent of semicon large scale integrated circuit techn semiconductor memories have 'a replaced the core memories.

Main memory of a computer is org as a number of storage 'locations', location being capable of storing one nu

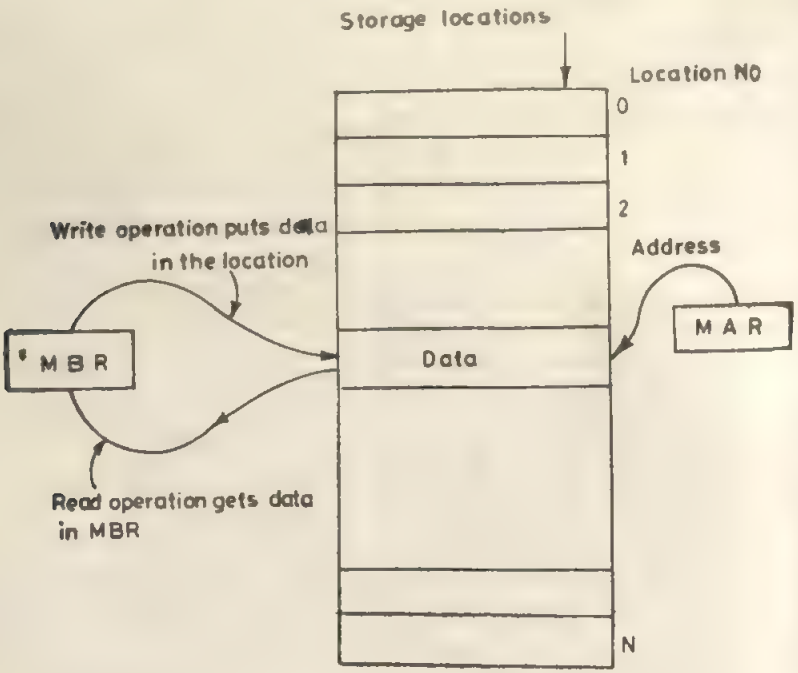



FIG 1·1 MAIN MEMORY SCHEMATIC

The access to the required location is provided by assigning another number, called address of the location. Main memory has register called memory address register (MAR) to hold the address of the location to be accessed. The memory also has an input-output port through which the data transfer takes place to and from the addressed location. This port is known as Memory Buffer Register (MBR) or Memory Data Register (MDR). Fig. 1.1 shows the schematic of a main memory.

### 1.2 3. Secondary Memories

These are storage devices which have very high storage capacity. Magnetic tapes, drums and disks are the commonly used devices for this purpose. These devices can store large amount of data, but the access methods are complex and require considerably more time compared to that of the main memory. Another feature is that, the basic unit of data transfer is a block (number of data items) instead of individual items. The recording principle here is the magnetisation of magnetic material due to the electric currents generated in accordance with the data being recorded.

### 1.2.4. Input/Output (I/O) Devices

The I/O devices are used for data transfer between a user and a computer system. Most commonly used input devices are card readers and paper tape/magnetic tape devices. A user punches his program and data on cards (or a paper tape) and these are fed to the device which recognises the punchings on cards/tape and transmit this data to the CPU or main memory in the acceptable form. Similarly results are printed on a printer. Teleprinters or CRT terminals are also commonly used in 'time sharing systems' where a number of users can be interactively served simultaneously by the computer system. These devices allow 'on-line' communication between users and computer system.

The I/O devices are connected to the computer through their controllers. A device controller initiates a device for I/O operation and provides the necessary control to carry out the effective communication between the computer and the device.

### 1.2 5. I/O Channels

In early computers, data transfer between the main memory and the I/O devices used to be carried out by the CPU itself. The I/O devices being considerably slower than the CPU, the overall 'throughput' thus will be limited, due to the waste of the processor time in waiting for the completion of an I/O operation. To avoid this, later computers are provided with channels to look after the I/O operations. In some systems special processor/processors are assigned for the purpose of handling I/O operations. These processors are called peripheral processors or I/O processors.

## 1 3. SOFTWARE COMPONENTS

Bare hardware components of the system cannot be used, unless, the system is provided with some supporting programs. These programs provide better communication interface between a user and system, without which machine will have to be programmed in its machine language ; thus restricting its use only to the users who can do machine language programming. The following software programs are commonly available which are introduced in this section.

- *Assembler
- *Compiler
- *Operating System
- *Application Programs.

### 1.3.1. Assembler

Basic instructions and data upon which a processor operates are coded as numbers. Therefore, to write a program for solving a problem (without any machine aid) user has to write programs constituted of series of numbers. Such a coding of problem 'algorithm' is cumbersome and thus will restrict

users only to those persons who can code the problem in machine language. To aid users, the first step in system software was the development of assembly languages. An assembly language allows the coding of programs with the help of names (mnemonics) instead of numbers. These mnemonics are used in place of numbers to indicate uniquely the operations and operands (addresses). For example, a machine language instruction say 051000 indicating the addition operation on the 'accumulator' and a variable X in the location 1000 may be written as ADD X. Assembly language programs are thus easier to write and understand than machine language programs.

A program written in the assembly language cannot be executed by the processor directly, since it can execute only its machine language program. But if it is translated into the machine language equivalent then the CPU can execute the translated program. Who will do this translation ? Well, computer itself could be programmed (may be in its machine language) to carry out this task. This program is called Assembler and is usually supplied by the manufacturer. Modern Assemblers not only provide the user with the symbolic language, but also allow more instructions, (user defined) in the assembly language than available on a bare (hardware) machine. These are called macros. A macro instruction when assembled (translated) is composed of a number of machine instructions which jointly carry out the operation of the macroinstruction.

### 1.3.2. Compilers

The ability of a computer as a translator to translate the symbolic language (Assembly) into machine language, led to the universal use of computers. This was made possible by providing various languages suitable for solving various classes of problems. For example, FORTRAN language is available for scientific computations, while COBOL language is used for business data processing. These languages are called higher level programming languages. A program written in a higher level language can be translated into a machine language by a special program. Such programs are called compilers. A particular higher level language could be made available on a computer installation by having a compiler for that language. Manufacturer of a system usually supplies compilers for a number of languages along with the system.

### 1.3 3. Operating System

A modern computer installation has a number of hardware and software elements. To communicate with these and utilise them efficiently, master control programs are used. These programs fall under the category of operating system. The basic task of an operating system is to accept user jobs in any of the available languages, schedule them for translation and execution and allocate them the required resources (memory, system programs, I/O devices etc.,) An operating system has a special language (called job control language) through which an user may specify the steps required for the execution of his job.

### 1.3.4. Application Programs

The software support discussed earlier gives user an easy interface for communication to the machine. All such software is categorised as system software. The application software on the contrary saves programming efforts of users by making available, a number of programs pertaining to their applications. Some systems may not have any such software support while others may have a number of application programs. For example, application programs like scientific subroutine package (SSP) and linear programming (LP) packages are available on IBM 360/370 series of machines.

## 1 4. HYPOTHETICAL COMPUTER : AN EXAMPLE

### 1.4.1. Organisation

In this section, we discuss a small hypothetical computer to bring out the basic

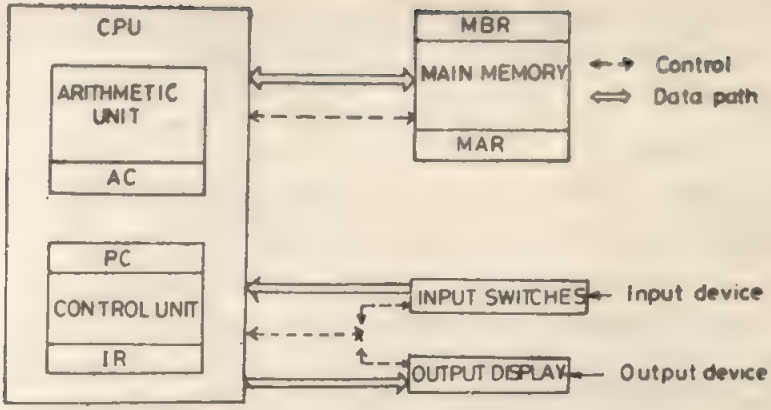



FIG. 1·2 BLOCK DIAGRAM OF A HYPOTHETICAL COMPUTER

principles involved in understanding and programming of a computer. We assume that the machine under discussion has 1000 locations (words) in main memory. Also we assume that each location can store 4-digit decimal numbers with sign. The block diagram of the machine is shown in Fig. 1.2.

The main memory stores data and instructions (program) required to solve a problem. Since a computer basically operates only on numbers, we must code every thing which is required to be told to the computer in terms of numbers. The first task at hand is to know, how the instructions of the machine are coded. For the hypothetical machine under discussion, we code an instructions using one word (4-digit signed decimal number) of the machine. The instructions has the following format.

| ± | 1 digit | 3 digits |
|---|---|---|
| Operation Code | | Address of the Operand |

Here 3 digits give the address of the main memory location where operand may reside. Since we assumed 1000 locations, 3 digits in the operand address field are sufficient to indicate any of the locations 000, 001, ..... 998 and 999. The most significant digit with sign in the instruction format indicates the operation to be performed (different one digit numbers with + or — sign shall mean different operations).

The operations are carried out on the operands in Accumulator (AC) register (accumulator register accumulates results and hence the name) and a location in main memory. In most cases the result of the operation is put back in the accumulator. We introduce now the machine language of this machine.

### 1.4.2. Machine Language

The machine language of the hypothetical machine has the following instructions :

In the list of instructions with operation codes (symbolic names given in brackets) presented next, an instruction word besides having an operation code will also have a 3-digit operand address indicating the memory location. In cases where two operands are involved in the operation, one of them shall be the accumulator, while the other operand shall be the memory location. The result of operation (involving two operands) will be left in the accumulator. For example, the instruction word +2210 means the addition of the numbers in the accumulator and the memory location 210. The result of this addition is put back in the accumulator.

+0 No Operation (NOP)
+1 Load Accumulator (LDA)
+2 Add to Accumulator (ADD)
+3 Multiply (MPY)
+4 Jump (JMP)
+5 Jump on Zero (JMPZ)
+6 Jump on Positive (JMPP)
+7 Read (READ)
+8 } Not defined at present
+9 } but can be defined and used

−0 Stop (STOP)
−1 Store Accumulator (STA)
−2 Subtract (SUB)
−3 Divide (DIV)
−4 Illegal (not to be used)
−5 Jump on not zero (JNZ)
−6 Jump on Negative (JMPN)
−7 Write (WRITE)
−8 } Not defined at present
−9 } but can be defined and used.

## HYPOTHETICAL MACHINE INSTRUCTION SET

The instructions (program) are stored in the memory. They are executed in sequence starting from the first instruction. Which instruction is to be executed next is told by the register named as program counter (PC) also called as the instruction address register (IAR). If the break in the sequence of execution of instructions is required, we execute a jump instruction which places the jump address in the PC. For example, the execution of the instruction +4205 (JMP 205) changes the program counter to the value 205. Conditional jumps are usually required and in this machine we have, JMPZ, JMPP and JMPN instructions, which can carry out conditional jumps under zero, positive and negative value of a number in the accumulator.

A {machine language program already present in the main memory is executed by placing the start address (address of first instruction) in the program counter and starting the computer (may be by pressing a start button). CPU carries out the following steps automatically to execute a machine language program.

### Steps

1. CPU transfers the contents of program counter to the MAR and reads the instruction into Instruction Register (IR).

2. CPU increments the Program Counter by one.

3. CPU executes the instruction as per the code. This may involve bringing out the operand from main memory. (Arithmetic unit provides the necessary facility to carry out arithmetic operations).

4. CPU goes to step 1 to execute the next instruction.

We illustrate the machine language programming principles by developing a program to add N numbers (assume that sum of these will not exceed +9999). Before writing the machine language program, we must develop an algorithm which precisely indicates how to go about doing the job. The algorithm is given in the flow-chart shown in Fig. 1.3. Each box in the flow-chart can be coded with the help of machine language instructions just discussed. The program is as follows :

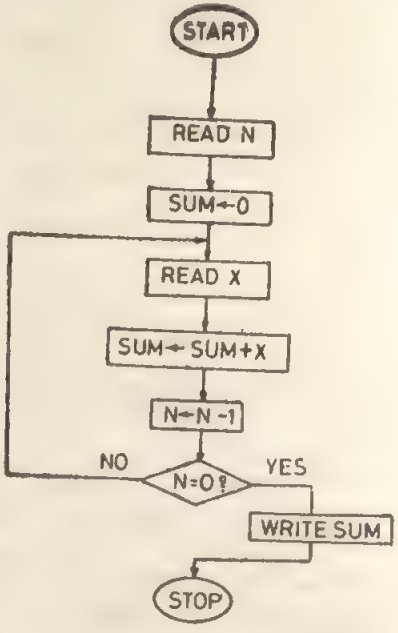



FIG.1·3 ALGORITHM FOR ADDITION OF NUMBERS

| *Machine Language Program* | | *Assembly Language Program* | | | *Comments* |
|---|---|---|---|---|---|
| *Location No.* | *Instruction/Data Word* | *Label* | *Mnemonic Code* | *Operand* | |
| 010 | +7023 | | READ | N | Read value of N |
| 011 | +1024 | | LDA | ZERO | S←0 *i.e.*, |
| 012 | −1025 | | STA | SUM | make SUM=0 |
| 013 | +7026 | LOOP | READ | X | Read the next number |
| 014 | +1025 | | LDA | SUM | SUM←SUM+X |
| 015 | +2026 | | ADD | X | |
| 016 | −1025 | | STA | SUM | |
| 017 | +1023 | | LDA | N | N←N−1 |
| 018 | −2027 | | SUB | ONE | |
| 019 | −1023 | | STA | N | |
| 020 | −5013 | | JNZ | LOOP | go to location 013 (LOOP) if AC≠0 |
| 021 | −7025 | | WRITE | SUM | Display SUM |
| 022 | −0000 | | STOP | | |
| 023 | | N | DL | | Declare a location for N |
| 024 | +0000 | ZERO | DC '0' | | Declare a constant ZERO |
| 025 | | SUM | DL | | Declare a location for SUM |
| 026 | | X | DL | | Declare a location for X |
| 027 | +0001 | ONE | DC '1' | | Declare a constant ONE. |

The above program is written in the machine language with instructions starting from location 010. Every instruction is also described (coded) in the mnemonic (assembly) language. The label (for example LOOP) in assembly language of the instruction or data corresponds to the memory location, where the instruction or data resides. It is the description of a location by a mnemonic name.

It is to be noted that different machines (computers) will have different machine (assembly) languages. A commercially available computer has many more instructions (than what we had with the hypothetical machine) varying from 50 in small computers to more than 200 in large machines.

## EXERCISES

1. CPU executes machine language program automatically. Can you suggest what microactivity (sub steps) CPU will have to carry out to execute a machine language instruction ?

2. In the example of machine language program discussed, the equivalent assembly language program was also written side by side. What additional information do you get from the assembly program ?

3. Which language will you choose to write programs, assembly or machine language ?

4. (*a*) Write a program in the assembly language of hypothetical machine to evaluate the expression

$T=a^3.b.c+d$, where $a$, $b$, $c$, $d$ and T are integers.

(*b*) Translate the program written for above in the machine language.

5. Repeat (4) for :

$T=a^n.b.c+d$ ($n$ is an integer)

6. Without using a MPY (multiply) instruction write a program to multiply two integers.

7. Repeat (6) for division. (Here division instruction should not be used).

8. Write a machine language program to add 100 integer numbers available as an array of 100 integers stored in consecutive memory locations.

9. An array of 100 integers is given. Write a program which will compact the array by moving O's to the bottom of the array.

10. Write a program in machine language to find the GCD (greatest common divisor) of two integers.

11. Repeat (10) for LCM (Least Common Multiple).

12. In the 2nd century A.D., Russian farmers invented an algorithm to carry out the multiplication of integers (grains or small stones were used to describe the integers). They could halve or double an integer number, and also they could add two integers (of course using stones). The multiplication algorithm is described below through an example of multiplication of X=25 by Y=35 (the method is called doubling and halving).

| | *Initial value* | | | | |
|---|---|---|---|---|---|
| Halve X | 25 | 12 | 6 | 3 | 1 |
| Double Y | 35* | 70 | 140 | 280* | 560* |

To obtain the product of X and Y, take the sum of numbers in the Y row for which the corresponding numbers in the X row are odd (*is shown on them). The product is :

$35+280+560=875$

Write a machine language program for the above multiplication procedure.

□

# 2

# NUMBER SYSTEMS

IN digital systems like digital computers, digital instrument systems, digital controllers etc., it is necessary to perform counting and arithmetic operations on numbers. Although the decimal number system is convenient for manual calculations, its usefulness in machine computation is limited due to the nature of electronic devices currently available. In most of the present day machines, numbers are represented and arithmetic operations performed in a system called binary number system. Also there are machines, which employ other number systems, but internally numbers are 'coded in a binary form'. This chapter is devoted to the study of number systems in general.

## 2.1. NUMBER REPRESENTATION

An ordinary unsigned decimal number actually represents a polynomial in 10. For example,

$$389{\cdot}48 = 3\times10^{2}+8\times10^{1}+9\times10^{0}+4\times10^{-1}+8\times10^{-2}.$$

Any unsigned decimal number thus could be written as :

$$N=\sum_{i=-m}^{n-1} a_i\times10^{i}$$

where $\sum$ denotes summation and $m$, $n$ are integers designating the number of digits in fractional and integral parts of N.

This representation of numbers is called 'decimal representation' and the number system is called the decimal number system since the number 10 is being used as a base. We may think of a generalised number representation whose base (or radix) is any positive integer. Consequently, any number N can be expressed as :

$$N=\sum_{i=-m}^{n-1} a_i\times r^{i}, \qquad ...(2.1)$$

where $a_i$ is a digit in the number system and its value is taken from the set $\{0, 1, 2...r-1\}$ and $r$ is any integer $> 1$.

The number N has $n$ digits in the integral part (left of the radix point) and $m$ digits in the fractional part (right of the radix point). The digit $a_{-m}$ is referred to as least significant digit (LSD), on the other hand $a_{n-1}$ is called 'most significant digit' (MSD). Due to the various weightages ($r^i$) given to the digits, these number systems are called the *positional* number systems. For example, the number 1011·01 in the binary number system ($r=2$) represents a polynomial in 2 as shown below :

MSD—↓ ↓—LSD

$$1011{\cdot}01 = 1\times2^{3}+0\times2^{2}+1\times2^{1}+1\times2^{0}+0\times2^{-1}+1\times2^{-2}$$

Most significant and least significant digits are marked as shown. The MSD has a weightage of $2^{n-1}$, while LSD has a weightage of $2^{-m}$.

Hereafter a number N in a base $b$ will be designated as $N)_b$ if the base of the number is not obvious from the context. Also all numbers we are talking about are assumed to be non-negative unless otherwise stated. For example, $128)_{10}$ stands for a number whose value is 128 in decimal representation.

Several number systems are possible by

TABLE 2.1 : Number systems and symbols commonly used for their digits.

| *Number System* | *Radix* | *Digits* 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hexadecimal | 16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| Duodecimal | 12 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | | | | |
| Decimal | 10 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |
| Octal | 8 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | | | | | | |
| Quinary | 5 | 0 | 1 | 2 | 3 | 4 | | | | | | | | | | | |
| Quoternary | 4 | 0 | 1 | 2 | 3 | | | | | | | | | | | | |
| Ternary | 3 | 0 | 1 | 2 | | | | | | | | | | | | | |
| Binary | 2 | 0 | 1 | | | | | | | | | | | | | | |

taking different integer values of $r$ in Equ. (2.1). Our discussion in this chapter is limited to values of $r \geq 2$ (number systems with negative radices had been studied by some research workers to exploit certain useful properties regarding the arithmetic). Table 2.1 presents the names given to the various number systems and the commonly used symbols of their digits.

Among these number systems, the knowledge of octal, hexadecimal, decimal and binary systems is essential in studying digital computing machines. Relevance of hexadecimal and octal representation is apparent when particularly large binary numbers are to be documented (as a part of computer systems documentation). For example a 36 bit binary number given below could be represented by very short 'string' of octal or hexadecimal digits.

**Binary Number :**
010010011101011110110001010100111111
**Hexodecimal Number** : 49D7B153F
**Octal Number** : 223536612477.

Thus the above binary number is represented by comparatively very short 'strings' of digits 49D7B153F in hexadecimal or 223536612477 in octal number system.

This is possible due to the fact that a digit in the octal or hexadecimal system has exactly $2^k$ ($k$ is an integer $\geq 1$) states. Therefore, we could group each $k$ bits (3 in case of octal and 4 in case of hexadecimal) starting from the radix point, and assign an equivalent digit from octal or hexadecimal system as the case may be.

## 2.2. CONVERSION OF NUMBERS

Suppose some number N, which we wish to express in $b_2$, is presently expressed in $b_1$. In converting a number from one base $b_1$ into another base $b_2$, it is convenient to distinguish two cases $b_1 < b_2$ and $b_1 > b_2$. Nevertheless any of the methods discussed below could be used.

**Case I $b_1 < b_2$ :** In this case the number is available as a polynomial in $b_1$ and since $b_1 < b_2$ we can employ $b_2$ arithmetic to evaluate the polynomial.

**Example 2.1** : Express numbers $462{\cdot}2)_8$ and $1101{\cdot}1)_2$ into decimal system.

$$\begin{aligned} 462{\cdot}2)_8 &= 4{\cdot}8^2 + 6{\cdot}8^1 + 2{\cdot}8^0 + 2{\cdot}8^{-1} \\ &= 256 + 48 + 2 + 2/8 \\ &= 306{\cdot}25)_{10} \\ 1101{\cdot}1)_2 &= 1{\cdot}2^3 + 1{\cdot}2^2 + 0{\cdot}2^1 + 1{\cdot}2^0 + 1{\cdot}2^{-1} \\ &= 8 + 4 + 1 + 1/2 \\ &= 13{\cdot}5)_{10}. \end{aligned}$$

In both the cases $b_2 = 10$ arithmetic has been used to evaluate the polynomials.

**Case II $b_1 > b_2$ :** Now it is more convenient to use base $b_1$ arithmetic. The conversion process is proved and explained by considering

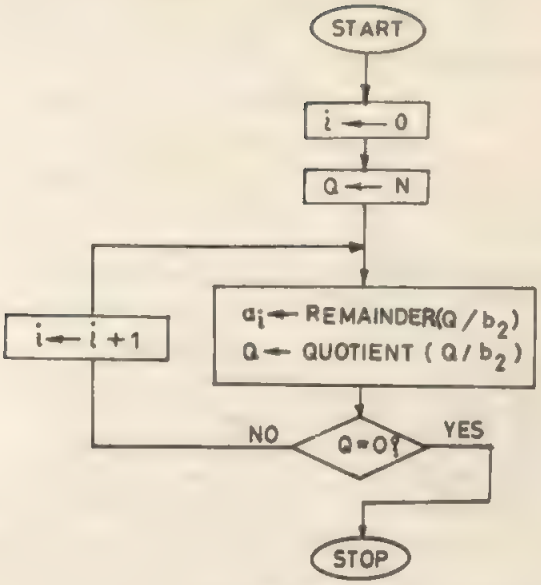



FIG. 2·1 CONVERSION OF INTEGERS

integral and fractional parts of a number separately.

**Integers** : Let $N)_{b1}$ be an integer whose value is available in the base $b_1$. Finding its representation in base $b_2$ is equivalent to finding the coefficients $a_i$s of the following polynomial.

$$N)_{b2}=a_{n-1}\, b_2^{n-1}+a_{n-2}\, b_2^{n-2}+\ldots a_1 b_2+a_0.$$

$N)_{b1}$ and $N)_{b2}$ are same numbers hence,

$$\frac{N)_{b1}}{b_2}=\frac{a_{n-1}\, b_2^{n-1}+a_{n-2}\, b_2^{n-2}+\ldots a_1 b_2+a_0}{b_2}$$

The results of this division are quotient Q and remainder R. Clearly $R=a_0$ because $a_0<b_2$ ($a_0$ is a digit in radix $b_2$) and $Q=a_{n-1}\, b_2^{n-2}+a_{n-2}\, b_2^{n}+\ldots a_1$. Dividing Q further by $b_2$ will similarly give $a_1$. If this process is repeated till Q=0, we shall obtain all digits of N in $b_2$. $a_0$ is the digit obtained at the very first step, $a_1$ at the next step and so on. A flow chart of this process is given in Fig. 2.1 and the process is summarized by the algorithm 2.1.

**Algorithm 2.1**

| *Steps* | *Comments* |
|---|---|
| 1. $i \leftarrow 0$ | Digit index set to zero, initial quotient is made equal to N. |
| 2. $Q \leftarrow N$ | |
| 3. $a_i \leftarrow$ Remainder $(Q/b_2)$ | Remainder of $Q/b_2$ is assigned to digit $a_i$ |
| 4. $Q \leftarrow$ Quotient $(Q/b_2)$ | Quotient of $Q/b_2$ is assigned to Q. |
| 5. If Q=0 then go to step 8 | |
| 6. $i \leftarrow i+1$ | |
| 7. Go to step 3 | |
| 8. Stop. | |

**Example 2.2** : Convert $326)_{10}$ into octal and binary

Decimal to octal conversion : $b_1=10$ and $b_2=8$

| Start | Q | R |
|---|---|---|
| Divided by | 326 | |
| 8 | 40 | 6 least significant digit |
| 8 | 5 | 0 |
| 8 | 0 | 5 most significant digit |

Q=0 hence stop

Decimal to Binary : $b_1=10$ and $b_2=2$

| Start | Q | R | |
|---|---|---|---|
| Divide by | 326 | | |
| 2 | 163 | 0 | Least significant bit |
| 2 | 81 | 1 | |
| 2 | 40 | 1 | |
| 2 | 20 | 0 | |
| 2 | 10 | 0 | |
| 2 | 5 | 0 | |
| 2 | 2 | 1 | |
| 2 | 1 | 0 | |
| 2 | 0 | 1 | Most significant bit |
| Q=0 hence stop | | | |

Thus $326)_{10}=506)_8$ or $101000110)_2$.

It is easy to convert an octal number into binary. We have to just substitute for each octal digit its respective binary equivalent (which normally could be remembered by every one of us). For example $506)_8$ is $101000110)_2$. Therefore, any conversion involving $b_2=2$ could also be achieved by converting initially to octal (or also hexa-decimal) and then replacing the digits obtained by their binary equivalents. This will result in drastic reduction of conversion steps.

**Fractions :** Let N be a fractional number in a base $b_1$. Let $N)_{b2}$ be its equivalent in a base $b_2$. Then we have :

$$N)_{b1}=a_{-1}b_2^{-1}+a_{-2}b_2^{-2}+\ldots a_{-m}b_2^{-m}.$$

The conversion of N into base $b_2$ is equivalent to finding the coefficients $a_{-1}$, $a_{-2}$, etc. If we multiply $N)_{b1}$ by $b_2$ we have :

$$N)_{b1}\times b_2=a_{-1}+a_{-2}b_2^{-1}+\ldots a_{-m}b_2^{-m+1}$$

It is obvious that $a_{-1}$ (being a digit) is an integer, while the rest of the expression above is a fraction (since every term has some negative power of $b_2$ and $a_i<b_2$).

$a_{-1}$ is actually the first (MSD) digit of the fraction N. If we take the fraction $a_{-2}b_2^{-1}+\ldots a_{-m}b_2^{-m+1}$ and repeat the multiplication by $b_2$, we shall get $a_{-2}$ and so on. Usually the process is repeated till the sufficient number of digits in the base $b_2$ are found because this process may never end in some cases. This is likely because a fraction with finite number of digits in one system may require infinite number of digits in some other system, for example, a number $0{\cdot}1)_8$ requires infinite digits in decimal system $(0{\cdot}3333\ldots)$.

The procedure of conversion of fractions is summarised by Algorithm 2.2 and Fig. 2.2

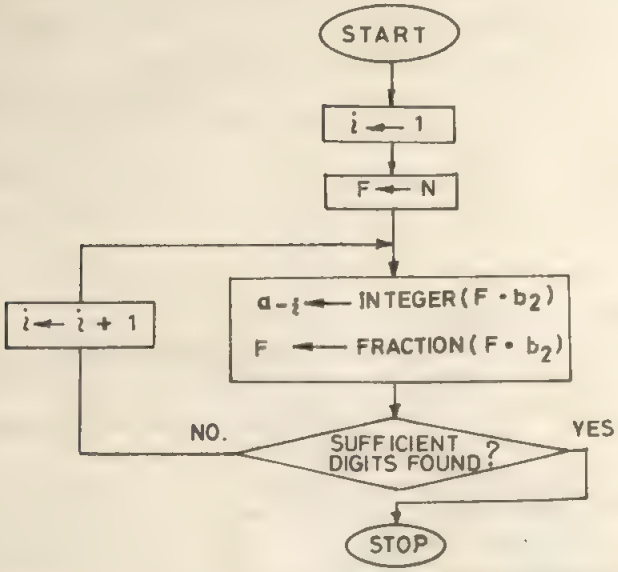



FIG. 2·2 CONVERSION OF FRACTIONS

represents its flow chart.

**Algorithm 2.2 : Conversion of fractions**

| *Steps* | *Comments* |
|---|---|
| 1. F←N, $i$←0 | Initial fraction is assigned to F, digit index set to 0 |
| 2. $i$←$i$+1 | |
| 3. $a_{-i}$←integer part of ($Fxb_2$) | Extract integer part of $Fxb_2$ and assign it to current digit |
| 4. F←Fractional part of ($Fxb_2$) | Assign new value to fraction |

5. If sufficient digits not found then go to the steps 2
6. Stop.

**Example 2.3 :** Convert 0·875 into binary

| binary digits | Fraction multiply by $b_2$=2 |
|---|---|
| MSB 1 | ·875×2 |
| 1 | ·750×2 |
| | ·50×2 |
| LSB | ·00 |

Result $0{\cdot}875)_{10}=0{\cdot}111)_2$

**Example 2.4 :** Convert $432\,354)_{10}$ into octal

(*i*) Integral part conversion

Divide by :

| | | |
|---|---|---|
| 432 | Reminder | |
| 54 | 0 | LSB |
| 6 | 6 | |
| 0 | 6 | MSB |

Result $660)_8$

(*i*) Fractional part conversion

| | |
|---|---|
| | 0·354×8 |
| 2 | ·832×8 |
| 6 | ·656×8 |
| 5 | ·248×8 |
| 1 | ·984 |

Result $0{\cdot}2651)_8$

The decimal number 432 354 is 660·2651 in octal system calculated up to four digits beyond the radix point.

## 2 3. COMPLEMENTS

**Definition 2.1** : The r's complement denoted by $d'$ of a digit $d$ in a number system with a radix $r$ is given by :

$$d'=\begin{cases} r-d & \text{if } d\neq 0 \text{ and} \\ 0 & \text{if } d=0. \end{cases}$$

For example, the 10's complement of '0' in decimal system is 0, while that of 6 is 4.

**Definition 2.2** : The $(r-1)$'s complement denoted by $d''$ of a digit $d$ in a number system with radix $r$ is given by the positive difference between $d$ and the largest digit in the number system.

For example, $(10-1)$'s *i.e.*, 9's complement in the decimal system of a digit 3 is $9-3=6$.

**Definition 2.3 :** The $r$'s complement denoted by N′ of a number N in a system with radix $r$ is given by $r^n-N$, and the $(r-1)$'s complement denoted by N″ is given by $r^n-N-r^{-m}$, where $n$, $m$ are number of digits in the integral and fractional parts of N.

**Example 2.5 :** Find $r$'s and $(r-1)$'s complement for the following numbers in the radices as indicated.

(*i*) $1011{\cdot}10)_2$ (*ii*) $983{\cdot}610)_{10}$ (*iii*) $563{\cdot}7)_8$.

(*i*) N=1011·10 has four digits in integral parts and 2 digits in the fractional part.

N′(2's complement$=2^4-1011{\cdot}10)_2=0100{\cdot}10$

and N″(1's complement$=2^4-1011{\cdot}10)_2-2^{-2}$,

$=0100{\cdot}01$.

(*ii*) $N=983{\cdot}610)_{10}$

N′(10's complement)$=10^3-983{\cdot}610$

$=016{\cdot}390$

N″(9's complement)$=10^3-983{\cdot}610-10^{-3}$

$=016{\cdot}389$

(*iii*) $N=563{\cdot}7)_8$

N′(8's complement)$=8^3-563{\cdot}7)_8=214{\cdot}1$,

N″(7's complement)$=214{\cdot}0$

### 2.3.1. Algorithms for r's and (r−1)'s Complements

The calculations required to find $r$'s and $(r-1)$'s complements (subtraction of $n$ digit numbers) can be simplified such that digit by digit complementing procedure could be employed. We are presenting the procedures for this in the following three algorithms.

**Algorithm 2.3 : (r−1)'s complement of a number N),**

Take $(r-1)$'s complement of all the digits.

**Algorithm 2.4 : r's complement of a number N),**

*(i)* Find $(r-1)$'s complement by algorithm 2.3.

*(ii)* Add $r^{-m}$, *i.e.*, add 1 to the least significant position

**Algorithm 2.5 : r's complement of a number N),**

| *Steps* | *Comments* |
|---|---|
| *(i)* $i \leftarrow -m$, F$\leftarrow$0 | F is a flag indicating first occurrence of a non-zero digit $d_i$, $i=1, 2...$ are digits of N. |
| *(ii)* $d_i \leftarrow d''_i$ | |
| *(iii)* If $d_i \neq 0$ then F$\leftarrow$1 | |
| *(iv)* $i \leftarrow i+1$ | |
| *(v)* If $i > n$ then go to step *(viii)* | |
| *(vi)* If F=1 then $d_i \leftarrow \bar{d}_i$ else $d_i \leftarrow d''_i$ | |
| *(vii)* Go to step *(iii)* | |
| *(viii)* Stop. | |

Algorithm 2·5 can be summarised as follows :

Scan the digits of the number from least significant digit till a non-zero digit say $d_k$ is found. Take $r$'s complement of all digits up to $d_k$ (including $d_k$), and $(r-1)$'s complement of the rest of the digits. The number thus formed will be the $r$'s complement of the original number.

**Example 2.6** : Find 10's complement of $936{\cdot}320)_{10}$

N=936·320

| | |
|---|---|
| Take 9's complement of the digits on this side, *i.e.*, 063 6 | Scan first non-zero digit Take 10's complement of the individual digits on this side, *i.e.*, 80 |

The result is N′=063·680.

**Example 2.7** : Find the 2's complement of a binary number 1011101000.

N=1011101000.

| | |
|---|---|
| Take 1's complement of the digits on this side 010001 N′=0100011000 | scan first non-zero digit Take 2's complement of the digits on this side. (*i e.*, leave them as they are) 1000 |

Note that the 2's complement of a binary digit is same as the digit itself and 1's complement is same as the Boolean complement of the truth value of a digit.

## 2.4. REPRESENTATIONS FOR SIGNED NUMBERS

In this section various representations of signed numbers are discussed.

### 2.4.1. Sign-Magnitude Form

Signed numbers are represented by two parts, *i.e.*, sign and magnitude. Magnitude part is represented by its value in the given number system, while the sign part is represented by digit value 0 to 1 (since sign can only be positive or negative). A '0' is usually assigned to the positive sign and a '1' is assigned to the negative sign. The two parts of the number may be separated by a comma for clarity.

**Example 2.8** : Number $+326)_{10}$ is represented by 0,326 while number $-326)_{10}$ is represented by 1,326.

**Example 2.9** : $+1101)_{10}$ and $-1011)_2$ are represented by 0,1101 and 1,1011 respectively.

The representation just discussed is called sign-magnitude representation. In this, we are using one additional digit position for the sign (though this digit takes either of the two values '0' or '1'). Now a question arises as to whether one can treat this digit in the polynomial of a number like other digits. If this is possible then sign digit will have a weightage of $r^n$, and the whole signed number could be treated on par with an unsigned number, and consequently design tasks would be simpler if one is designing a machine to carry out the arithmetic operations. This is not possible with the sign-magnitude representation. For example, a number $-33$ in decimal is represented as 1, 33. If this is treated as 133 ignoring the comma, it would mean a different number. Also suppose we assign $-10^n$ weightage instead of $10^n$ to the

sign digit, then the number represented would become $-1\cdot10^2+33=67$ which is also a different number. This drawback is removed by the representation discussed in the next paragraph, where positive and negative numbers are uniformly represented by a polynomial in $r$.

### 2.4.2. Sign-Complement Form

In this representation sign of a number is represented as discussed earlier while the magnitude part of a positive number represents its true value. On the other hand if the number is negative its magnitude part represents the complement of the actual value. The sign digit occupies $n^{th}$ position and its weight is assumed as usual to be $n^{th}$ power of $r(r^n)$ with a negative sign. This arrangement can now represent all signed numbers whether positive or negative, uniformly by the Equation 2.1. Hence the number could be treated as if it were an unsigned one. This is helpful in mechanising addition and subtraction of signed numbers.

**Example 2.10** : (*i*) $+356)_{10}$

$N=0, 356$

$=-0\cdot10^3+3\cdot10^2+5\cdot10+6$

$=-0\cdot1000+356$

$=+356.$

(*ii*) $-356)_{10}$

$N=1, \overline{356}$ (number is negative hence magnitude in complement form)

$=1, 644$

$=1, 644$ (with negative weight for MSD *i.e.*, sign)

$=-1\cdot10^3+6\cdot10^2+4\cdot10+4$

$=-1000+644$

$=-356.$

The above example has demonstrated that in decimal system, sign-10's complement representation uniformly represents negative as well as positive numbers. Hence we can include the sign digit in the magnitude part and forget that we had a signed number (except that MSD has a negative weight).

**Proposition 2.1** : A signed number from a system with radix $r$ could be represented in sign—$r$'s complement form, such that its value could be uniformly evaluated by a polynomial of Equation 2.1 with negative weightage for most significant digit.

**Proof : Case 1. Positive number N**

Sign —$r$'s complement representation for this number has sign digit '0' and N as the magnitude. The value of the combined number formed thus is $-0.r^n+N=N$.

**Case II : Negative Number N**

$-N$ has a representation 1, N', *i.e.*, $-1.r^n+(r^n-N)$ which is $=-N$.

Since there are two types of complements, we have two representations for signed numbers : (*i*) Sign —$r$'s complement representation and (*ii*) Sign—$(r-1)$'s complement representation. Unless otherwise stated complements of numbers throughout this book shall mean $r$'s complements.

This chapter has provided an uniform treatment of numbers in various number systems. The material could be utilised for any number system encountered by a computer science student. Examples were selected from binary, octal and decimal systems with the view to get him acquainted with these systems, which are commonly used in digital computers.

## EXERCISES

1. Convert the following numbers
   (*a*) $53\cdot1575)_{10}$ to base 2.
   (*b*) $1812)_{10}$ into octal and hexadecimal.
   (*c*) $1101\cdot110101100)_2$ into octal and hexadecimal.
   (*d*) $156723)_8$ into decimal
   (*e*) $1567)_9$ into ternary and duodecimal.

2 Solve the following equations for unknown base $b$

(*a*) $25)_{10}=27)_b$. (*b*) $263)_8=455)_b$
(*c*) $452)_{10}=318)_b$.

3. Given the octal numbers X=367·2, Y=36·12 and Z=171·1, perform the following operations

(*a*) X−Z (*b*) X+Y
(*c*) X/Y (*d*) X.Z

4. Repeat (3) for binary numbers X=1011·01, Y=101·11 and Z=0111·1.

5. Each of the following arithmetic operations is correct in at least one number system. Determine the possible bases in each operation.

(*a*) 41/3=13 (*b*) $\sqrt{41}=5$
(*c*) 44/4=11 (*d*) $13^3=11113$.

6. Determine the missing number of the series.

(*a*) 10000, 121, 100, ?, 24, 22, 20
(*b*) 101, 22, 32, 50, ?, 114, ...

7. Find the *r*'s and (*r*−1)'s complements for the following numbers in bases as indicated.

(*a*) $123456)_{10}$ (*b*) $12345)_6$
(*c*) $123456)_8$ (*d*) $1234567)_{16}$.

8. Find the 1's and 2's complements for the following binary number.

(*a*) 1011010 (*b*) 1101·011
(*c*) 0111000 (*d*) 00001

9. Represent the following signed numbers in sign-magnitude, sign (*r*−1)'s-complement and sign *r*'s-complement form.

(*a*) $+326)_8$ (*b*) $-367)_{10}$
(*c*) $+1010)_2$ (*d*) $-1101)_2$
(*e*) $-763)_8$

□

# 3

# BOOLEAN ALGEBRA AND COMBINATIONAL LOGIC DESIGN

BOOLEAN algebra of two elements provides the mathematical base for a number of applications. One of the major contributions to this application spectrum is the design of digital computers. In general, Boolean algebraic structures are possible with number of elements as powers of 2. In this chapter, a study and applications of two element Boolean algebra is presented. The terms Boolean, logic and switching are used synonymously in this book.

## 3.1. BOOLEAN ALGEBRA : AXIOMS

**Definition 3.1** : A Boolean algebra $B_2$ is the quadruple $\{B, +, ., '\}$ where B is the set $\{0, 1\}$ and operations '+' called OR (sum), '.' called AND (product) and ' (prime) called complementation satisfy the following axioms :

**Axiom 1** : There exists a variable $x$ such that

$x=0$ if only if $x \neq 1$

and $x=1$ if and only if $x \neq 0$.

**Axiom 2** : '+' (OR) and '.' (AND) are binary operations satisfying the following tables called truth tables :

AND Table

| $x$ | $y$ | $x.y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR Table

| $x$ | $y$ | $x+y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Axiom 3** : '(prime) is an unary operation called complementation (or negation or NOT) such that :

$x'$ is a complement of $x$ if and only if

$x+x' = 1$ and

$x.x' = 0$ holds for all $x$ belonging to set $B$.

Complement of $x$ is also denoted by $\bar{x}$ or NOT $x$.

### 3.1.1. Alternate Statements of Axiom 2

1. The elements '0' and '1' are sometimes referred to as FALSE and TRUE respectively. Then axiom 2 states that $x$ OR $y$ *i.e.*, $(x+y)$ is true if $x$ is true or $y$ is true ;

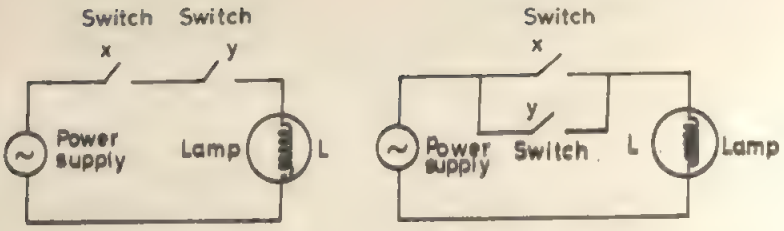



FIG. 3-1 ELECTRIC NETWORK ANALOG OF AND/OR

and $x$ AND $y$ is true if and only if $x$ is true and $y$ is true.

2. Yet another equivalent of axiom 2 is as follows. This definition will be useful in proving basic properties of Boolean algebra $B_2$. Let us order the set B by defining an ordering relation $<$ (less than). If $0<1$ in set B, then the following definitions are immediate consequence of relation $\leq$ (less than or equal to)

AND : $x.y=\min(x, y)$

OR : $x+y=\max(x, y)$

where min $(x, y)$ denotes the smallest of $x, y$ and max $(x, y)$ denotes the largest of $x, y$.

### 3.1.2. Electric Network Examples

The electric networks shown in Fig. 3.1 (*a*) and Fig. 3.1 (*b*) depict the meanings of the axiom 2.

Let ON and OFF states of the switches $x$, $y$ and lamp L be represented by the logical '1' and '0' respectively. Then it can be noted easily that :

In Fig. 3.1. (*a*) lamp is ON when (*i.e.*, $L=x.y$) both $x$ and $y$ switches are ON (AND operation analog).

In Fig. 3.1 (*b*) lamp is ON when $x$ is ON or $y$ is ON (*i.e.*, $L=x+y$).

## 3.2. PROPERTIES AND THEOREMS

With the help of axioms stated in section 3.1 it is possible to verify that the switching algebra (Boolean algebra) satisfies the following fundamental and important properties for all variables $x$, $y$ and $z$ belonging to the set B.

**Basic Properties**

| | | |
|---|---|---|
| 1. Commutative : | (*a*) | $x+y=y+x$ |
| | (*b*) | $x.y=y.x$ |
| 2. Associative : | (*a*) | $x+(y+z)=(x+y)+z$ |
| | (*b*) | $x.(y.z)=(x.y).z$ |
| 3. Distributive : | (*a*) | $x+y.z=(x+y).(x+z)$ |
| | (*b*) | $x.(y+z)=x.y+x.z$ |
| 4. Involution : | | $(x')'=x$ |
| 5. Idempotency : | (*a*) | $x+x=x$ |
| | (*b*) | $x.x=x$ |
| 6. | (*a*) | $1+x=1$ |
| | (*b*) | $1.x=x$ |
| 7. | (*a*) | $0+x=x$ |
| | (*b*) | $0.x=0$ |

These properties could be proved by constructing truth tables for both the sides of equalities. Alternate algebraic proofs could be given using max and min definitions of axiom 2 for OR and AND respectively. Hereafter, whenever no operator appears between two operands, a '.' may be assumed.

**Example 3.1 :** Prove the properties 5 and 6 (*a*)

Property 5 : $x+x=x$

**Proof :** $x+x=\max(x, x)=x$

$x.x=\min(x, x)=x.$

Property 6(*a*) : $1+x=1.$

**Proof :** $1+x=\max(1, x)$, '1' being the largest element in B, we have $\max(1, x)=1$, hence $1+x=1$.

The seven properties listed above are useful in simplifying logic expressions. It is the associative property of this algebra which allows us to define + and . operations on more than two variables.

**Example 3.2.** Prove the following equalities.

(*a*) $x+xy=x$ (Absorption)

(*b*) $x+x'y=x+y$

**Proofs :** (*a*) $x+xy=x.1+x.y$ (property 6 (*b*))
$=x.(1+y)$ (property 3 (*b*))
$=x.(1)$ (property 6 (*a*))
$=x$ (property 6 (*b*))

(*b*) $x+x'y=(x+x').(x+y)$ (property 3 (*a*))
$=1.(x+y)$ (axiom 3)
$=x+y$ (property 6 (*b*))

**Example 3.3 :** Simplify the expression T $=xy+x'z+xyz+x'yzw$

$T=xy+xyz+x'z+x'zwy$ (property 1)
$=xy\,(1+z)+x'z\,(1+wy)$ (property 3 (*b*))
$=xy\,.\,1+x'z.1$ (property 6 (*a*))
$=xy+x'z$ (property 6 (*b*))

For all $x$, $y$ and $z$ belonging to set B, following theorems also hold.

**Theorem 3.1 : Consensus Theorem**

(*a*) $xy+x'z+yz=xy+x'z$

(*b*) $(x+y)(x'+z)(y+z)=(x+y)(x'+z)$.

**Proof :** (*a*) $xy+x'z+yz=xy+x'z+yz.1$
$=xy+x'z+yz(x+x')$
$=xy+x'z+yzx+yzx'$
$=xy+xyz+x'z+x'zy$
$=xy(1+z)+x'z(1+y)$
$=xy+x'z.$

(*b*) $(x+y)(x'+z)(y+z)$
$=(x+y)(x'+z)(y+z+x'x)$
$=(x+y)(x'+z)(y+z+x')(y+z+x)$
$=(x+y)(x+y+z)(x'+z)(x'+z+y)$
$=(x+y)(x'+z)$

**Theorem 3.2. De Morgan's Theorem**

(*a*) $(x+y)'=x'.y'$

(*b*) $(x\,y)'=x'+y'$

**Proof :** (*a*) $(x+y)'=x'.y'$

Let $A=x+y$, and $B=x'.y'$

Hence $A+B=x+y+x'.y'$
$=x+y'+y$
$=x+1$
$=1,$

and $A.B=(x+y)(x'y')$
$=xx'y'+yx'y'$
$=0$

Therefore by axiom 3, B is a complement of A

Hence $A'=B$, substituting back $x+y$ and $x'y'$ for A and B respectively we have

$$(x+y)'=x'.y'$$

(*b*) This part follows from part (*a*)

*i.e.*, Let $A=x'$ and $B=y'$

then

$$A.B=x'y'$$

and

$$(A'+B')'=(x+y)'$$

We have proved that property (*a*) holds, therefore,

$(A'+B')'=A.B.$, by involution property, we have $(A.B)'=A'+B'$

**Example 3.4 :** Find the complement of the expression

$f=AB(C+D')+C+A'\,D$, and then simplify.

$f'=\overline{A.B.(C+D')+C+D.A'}$

Let $x=A.B.\,(C+D')$ and $y=C+D.A'$

Then $f'=\overline{x+y}=x'.y'$
$=\overline{AB\,(C+D')}\;.\;\overline{C+DA'}$
$=(\overline{AB}+\overline{(C+D')})\;.\;(C'.\,\overline{DA'})$
$=(A'+B'+C'D)\,.\,(C'\,.\,(D'+A))$
$=(A'+B'+C'D)\;.\;(C'D'+C'A)$

The new expression could be simplified as follows :

$f'=A'C'D'+B'C'D'+B'C'A+C'AD$
$=C'(A'D'+B'D'+B'A+AD)$

Using consensus theorem on terms $A'D'+B'D'+B'A$ we have

$f'=C'(A'D'+B'A+AD)$

The use of Boolean algebra in simplifying the complex sentences is demonstrated by the following example.

**Example 3.5 :** A hypothetical insurance company issues policies to applicants on the basis of the following conditions. An applicant must satisfy at least one of the conditions stated below so as to obtain a policy. Find the simplified policy issue statement.

1. Applicant is a married male of age 25 years or above.
2. Applicant is a female who never met with a car accident.
3. Applicant is a married female and has met with a car accident.
4. Applicant is a male below 25 years.
5. Applicant is not below 25 years and never met with a car accident.

Boolean algebra could be used to simplify the above complex policy statement. We assign Boolean variables to the elementry propositions contained in the above statements. They are as follows .

| Boolean Variable | : Elementry Proposition |
|---|---|
| M | : an applicant is married |
| S | : an applicant is a male |
| C | : applicant met with a car accident |
| Y | : applicant is below 25 years of age |
| P | : policy is to be issued. |

The above variables take true values if the respective propositions are true and the complemented variables shall assume true values if the respective propositions are false.

Therefore, now the above five statements can be symbolically written as below :

1. SY′M
2. S′C′
3. S′MC
4. SY
5. Y′C′

The policy is issued if any of the five statements are true about an applicant, thus

$$P=SMY'+S'C'+S'MC+SY+Y'C'$$

Simplification of this expression shall give us a simplified policy statement. So we now proceed to simplify the expression P.

$$\begin{aligned}P&=MSY'+YS+S'C'+Y'C'+MCS'\\&=S(Y+Y'M)+S'(C'+MC)+Y'C'\\&=S(Y+M)+S'(C'+M)+Y'C'\\&=YS+MS+S'C'+S'M+Y'C'\\&=YS+MS+MS'+S'C'+Y'C'\\&=YS+M+S'C'+Y'C'\\&=M+YS+C'(Y'+S')\\&=M+YS+C'(YS)'\\&=M+YS+C'\end{aligned}$$

Substituting the elementry propositions for each of the variables we have the following policy statement.

Policy is to be issued if :

1. applicant is married , or
2. applicant is a male below 25 years or
3. applicant has never met with a car accident.

## 3.3. CANONICAL FORMS OF SWITCHING FUNCTIONS

A given switching expression represents a switching function. Many expressions may stand for a single switching function, *i.e.*, all of the expressions may give exactly the same truth table. A switching function can be expressed by unique type of expressions called normal forms. There are two types of normal forms of a switching function, *viz.*, (*i*) canonical sum of products (also referred to as disjunctive normal form) and (*ii*) canonical product of sums (conjunctive normal form).

**Definition 3.2.** An appearance of a variable in the expression either in true or complemented form is called a literal.

For example terms $xy'z$ and $xy$ have three and two literals respectively.

**Definition 3.3.** A product term in a sum of products switching expression for a function of $n$ variables is called a minterm if it has exactly $n$ literals.

**Definition 3.4.** A sum term in a product

of sums expression for a function of $n$ variables is called a maxterm if it has exactly $n$ literals.

**Example 3.6.** Let $f_1(w, x, y, z)$
$= x'y + wxyz' + x'zy + w'x'y'z$
and $f_2(w, x, y, z)$
$= (x' + y) \cdot (w' + x' + y + z') \cdot (x' + z + y)$

The terms $w\,xyz'$ and $w'x'y'z$ are minterms (they have 4 literals each), whereas $x'y$ and $x'zy$ are not minterms.

In $f_2$, $(w' + x' + y + z')$ is a maxterm, while others are not.

**Proposition 3.1.** A minterm assumes value '1' for exactly one combination of variables.

Minterm being a product term (AND term with all variables present in either true or complemented form) will assume value '1' only when all the literals assume value '1'. For example, the minterm $wxyz'$ is '1' only for $w = x = y = 1$ and $z = 0$. Its value is 0 for rest of the possible values of $w$, $x$, $y$ and $z$.

**Proposition 3.2.** A maxterm assumes a value '0' for exactly one combination of variables.

Given a switching expression one may be required to find its canonical forms. The following subsections discuss this procedure.

### 3.3.1. Canonical Sum of Products

Pick up a term from the given sum of products expression. If it is a minterm leave it and take the another term. Find a missing variable say $x$ in the term. Multiply (AND) the term by $(x + \bar{x})$, (*i.e.*, '1'). This will not affect a value of the term picked up. Expand this product. This will give you two terms. Repeat this process till all the product terms contain a literal for each of the variables. The resulting expression is the canonical sum of products, (*i.e.*, an expression having sum of minterms).

**Example 3.7.** Find the canonical sum of products expression for

$$T = xy + x'y' + wx'z + x'y'z'w$$
$$T = xy(w + w') + x'y'(w + w') + wx'z + x'y'z'w$$
$$= wxy + w'xy + wx'y' + w'x'y' + wx'z + x'y'z'w$$
$$= wxy(z + z') + w'xy(z + z') + wx'y'(z + z') + w'x'y'(z + z') + wx'z(y + y') + x'y'z'w$$
$$= wxyz + wxyz' + w'xyz + w'xyz' + wxy'z + wx'y'z' + w'x'y'z + w'x'y'z' + wx'zy + wx'zy' + wx'y'z'$$
$$= wxyz + wxyz' + wx'zy + wx'zy' + wx'y'z' + w'xyz + w'xyz' + w'x'y'z' + w'x'y'z$$

The minterms merely represent combinations of input variables for which a function has a true value. Thus in Example 3.7, the combinations of $w$, $x$, $y$, $z$ for which the function has a value '1' are (1 1 1 1), (1 1 1 0), (1 0 1 1), (1 0 0 1), (1 0 0 0), (0 1 1 1), (0 1 1 0), (0 0 0 0) and (0 0 0 1). These are written merely by taking a term and writing a '1' for variable appearing in true form and '0' if a variable is in the complementd form. Sometimes a minterm is represented by the number formed by its literal combination taken in the predefined order. For example, the minterm $wxyz'$ of a function $T(w,x,y,z)$ of Example 3.7 is represented by 1 1 1 0. Hence this minterm may be represented by a number 14. With this notation a function of Example 3.7 could be written as $T(w,x,y,z) = \Sigma(0, 1, 6, 7, 8, 9, 11, 14, 15)$.

Note that we have only listed the minterms of T in the above summation and $\Sigma$ stands for sum (OR).

Knowing a 'sum of products' expression of a function is equivalent to knowing its truth table. Because, the minterms present in the expression give the combinations for which function has a value '1'. Hence, for rest of the combinations, it will have a value 0.

The truth table for the above function is now constructed using the minterms obtained. It is shown in Table 3.1.

TABLE 3.1. Function of Example 3.7

| $wxyz$ | *Minterms* | *Dec. Num.* | $f$ |
|---|---|---|---|
| 0000 | $w'x'y'z'$ | 0 | 1 |
| 0001 | $w'x'y'z$ | 1 | 1 |
| 0010 | | | 0 |
| 0011 | | | 0 |
| 0100 | | | 0 |
| 0101 | | | 0 |
| 0110 | $w'xy'z'$ | 6 | 1 |
| 0111 | $w'xyz$ | 7 | 1 |
| 1000 | $wx'y'z'$ | 8 | 1 |
| 1001 | $wx'y'z$ | 9 | 1 |
| 1010 | | | 0 |
| 1011 | $wx'yz$ | 11 | 1 |
| 1100 | | | 0 |
| 1101 | | | 0 |
| 1110 | $wxyz'$ | 14 | 1 |
| 1111 | $wxyz$ | 15 | 1 |

Suppose that a truth table of a switching function is given and we have to find the algebraic expression of a function represented by the table. The procedure though we have not studied yet, looks obvious, *i.e.*, collect all the combinations for which the function has a value '1', and carry out the following :

Write down the minterms for the combinations for which function value is 1.

Take a sum (OR) of these minterms.

The above procedure gives us the canonical sum of the products form of the function. This expression could be simplified if required and used.

**Example 3.8.** Write down the canonical sum of products expression for a function defined by the Table 3.2.

TABLE 3.2

| $x$ | $y$ | $z$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

The minterms of $f$ are (000), (010), (100), (101), and (110) *i.e.*,

$$x'y'z',\ x'yz',\ xy'z',\ xy'z,\ xyz'.$$

Therefore, the required expression is :

$$f(x, y, z)=x'y'z'+x'yz'+xy'z'+xy'z+xyz'$$

### 3.3.2. Canonical Product of Sums

Another canonical form of a switching function called canonical product of sums exists. The procedure to find it, is the dual of the one discussed in section 3.3.1. We collect all the maxterms of a function and write them as a product expression. A maxterm represents the combination for which a function has value '0'.

A maxterm is a sum term written such that a true variable appears only if the value of the variable is 0 in the combination. For example, combinations (0001), (0010) and (0000) represent the maxterms $(w+x+y+z')$, $(w+x+y'+z)$, $(w+x+y+z)$ respectively for a function of variables $w$, $x$, $y$ and $z$.

**Example 3.9.** Write the canonical product of sums expression for the function given by Table 3.2.

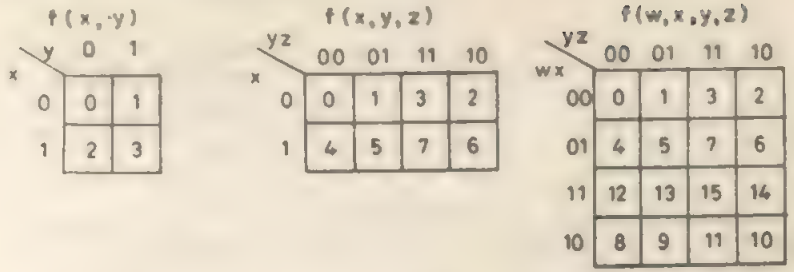



FIG. 3.2 Karnaugh maps showing addresses of squares.

The maxterms of $f(x, y, z)$ (see Table 3.2) correspond to the combinations (001), (011) and (111) (for which $f=0$). These are $(x+y+z')$, $(x+y'+z')$ and $(x'+y'+z')$, and the function in the canonical product of sums form is :

$$f=(x+y+z') \cdot (x+y'+z') \cdot (x'+y'+z').$$

Sometimes $f$ is represented by the numbers formed by the input combinations for which a function has value '0'. For example, the above function in the canonical product of sums form is represented as :

$$f(x, y, z)=\pi(1, 3, 7).$$

## 3.4. KARNAUGH MAPS AND SIMPLIFICATION OF BOOLEAN EXPRESSIONS

Use of properties and theorems of Boolean algebra to simplify expressions sometimes requires the great ingenuity on the part of a person who has to carry out simplification. This is because one may not foresee the use of an appropriate theorem or property at a particular step of a simplification procedure. The map method introduced by Karnaugh not only reduces the above hurdle but also provides a systematic procedure. This method is extremely popular and useful for manual simplifications of functions of upto six variables.

### 3.4.1. Karnaugh Maps

A 'Karnaugh map' is nothing but a truth table of a switching function arranged in a two-dimensional manner as an array of squares. Each square corresponds to an input combination of a function. Fig. 3.2 shows these maps for functions of 2, 3 and 4 variables. The numbers formed by input combinations are written in squares to indicate the addresses of the squares.

Usually these numbers are not to be written. A switching expression to be simplified is converted in the normal form, and the function values are entered in the respective squares. For example, if $f(w, x, y, z)$ has a form $\Sigma(1, 5, 8, 11, 13, 15)$, the squares marked 1, 5, 8, 11, 13 and 15 are filled with a value '1' and rest of them are filled with a '0'.

**Example 3.10.** Fill the Karnaugh map for a function

$$f=xy+x'z'+xyz$$

The canonical form for this function can be found using techniques discussed.

Thus $f=\Sigma(6, 7, 3, 1)$

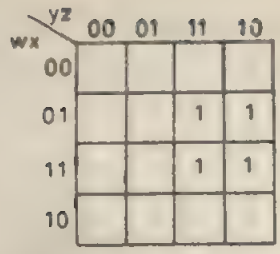
or $f=\pi(0, 2, 5, 4)$

This being a function of 3-variables, map having eight squares is produced below, which is filled using either of the normal forms as shown below :

| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

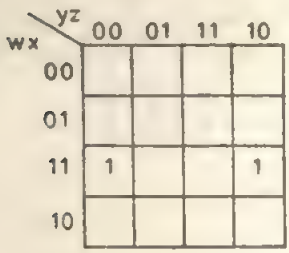
Another way to fill the map is as follows :

Pick up a product term and enter a value '1' (0 in case of sum term) for all the squares whose addresses satisfy truth values of variables present in the term.




Four squares are equivalent to xy.

FIG. 3.3(a)




Two squares shown are equivalent to z'wx.

FIG. 3.3(b)

**Example 3.11.** The terms $xy+z'wx$ of function $f(w, x, y, z)$ are filled as shown in Fig. 3.3. It can be noted that in Fig. 3.3 (*a*), $x$ and $y$ are constant at value 1 for all the four squares. Similarly in Fig. 3.3 (*b*), both the square have constant values 0, 1 and 1 for $z$, $w$ and $x$ respectively.

### 3.4.2. Simplifications of Expressions

Simplification of expressions is based on the following observations about the properties of expressions.

$xy'+xy=x$, *i.e.*, two terms (10 for $xy'$ and 11 for $xy$) differing only in the value of one variable can be combined to form a single term which represents a variable (or expression) with a constant value in these terms, *i.e.*, the function $f_1=xy'+xy'$ depends only on $x$ here. Values of $y$ do not matter (because $f=x$, for both the possibilities of $y$). To easily locate such terms, the rows and columns in the map are arranged such that this is true for every pair of neighbouring rows or columns. Extreme columns/rows are assumed to be neighbouring.

**Definition 3.5.** Two terms are said to be logical neighbours if they differ in the value of only one variable.

The numbering arrangement of rows and columns is such that logical neighbours are also physical (in the maps) neighbours. Therefore, we have just to combine physically neighbouring squares for the purpose of simplification.

**Combining Four Squares**

The Boolean expression for four neighbouring squares having same function value is of the form :

$$f_1=xyz+xy'z+xyz'+xy'z'$$
$$=x(yz+y'z+yz'+y'z')$$
$$=x(1).$$

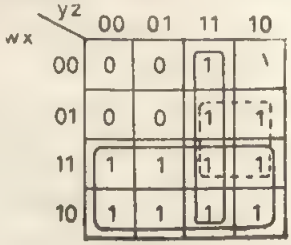
The four terms represent coordinates such that only $x$ value is constant while $y$, $z$ take all possible values. For example : squares representing $xyz$=110, 111, 101 and 100. If such squares are found on the map and have the same function entry, they could be combined to give a single term represented by the constant coordinate variable (*i.e.*, $x$ in this case).

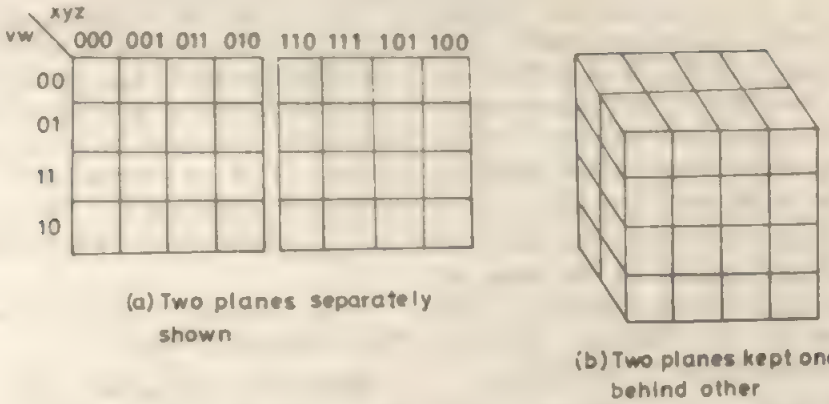
Similarly 8, 16, etc., squares could be combined. Combining $2^k$ squares eliminate k variables because k variables shall take all possible values for which function value is constant.

**Example 3.12.** Simplify the function

$f(w, x, y, z)=xy+wz+y'z'w+x'yz+wy$

The map for this function is shown below :






(a) Two planes separately shown

(b) Two planes kept one behind other

FIG.3·4 5-VARIABLE MAPS

A maximum possible number of squares are to be combined such that we eliminate maximum possible number of variables and at the same time cover as many '1's as possible which ensures lesser number of terms in the final simplified expression. The various groups of squares are marked by enclosing them.

In example 3.12 a group of eight squares, (bottom two rows) gives a term $w$ ($w$ is constant at 1 while other variables take all possible values) second group is given by column 11. These squares can be combined to give a term $yz(y=1, z=1$ while other variables take all possibilities). Similarly third group (enclosed by dotted line) gives us the term $xy$.

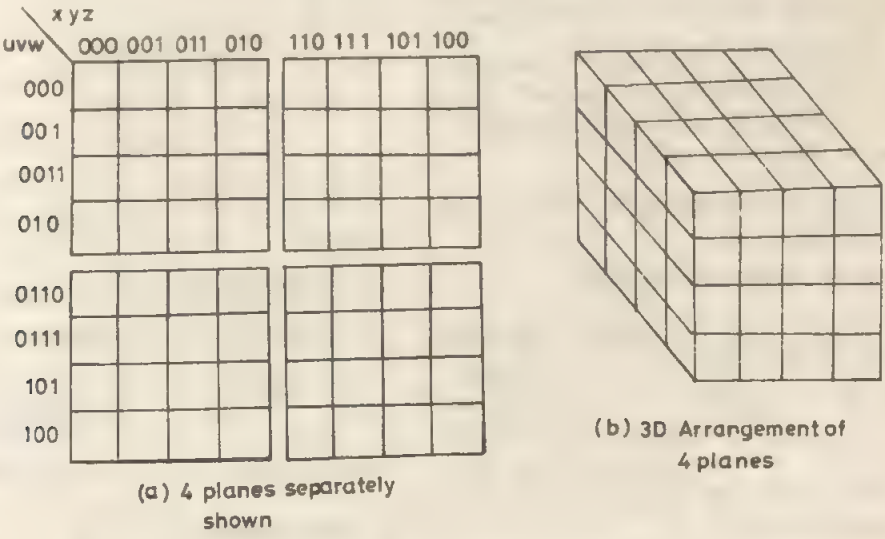
The simplified expression for $f$ is given by :

$$f=w+yz+xy$$

Note that in this example some of the squares have been used in more than one group. This does not affect the result due to the idempotent property of Boolean algebra (*i.e.*, $x+x=x$).

**Five and Six Variable Maps**

Maps having upto four variables could be




(a) 4 planes separately shown

(b) 3D Arrangement of 4 planes

FIG. 3·5 6-VARIABLE MAPS

arranged such that physical and logical neighbourhoods of squares are possible. For the maps of more variables it is difficult to get such an arrangement in a 2D planer map. However, this is possible if one can imagine a multidimensional map. These maps are shown in Fig. 3.4 (*b*) and Fig. 3.5 (*b*).

Note that for simplifying the functions, the neighbouring squares (in the sense of Fig. 3.4 and Fig. 3.5 can be combined. The same procedure as described for four variables could be employed here also.

## 3.5. FUNCTIONALLY COMPLETE SETS OF PRIMITIVES

**Definition 3.6.** A set of primitives (operations) is said to be functionally complete if and only if any arbitrary switching function could be expressed using these operations.

**Proposition 3.3.** A set of operations $\{ . , + , ' \}$ defined in setion 3.1 is functionally complete.

**Proof.** Follows from the discussion of canonical forms of functions in section 3.3.

**Theorem 3.3.** The sets of primitives $\{ . , ' \}$ and $\{ + , ' \}$ are functionally complete.

**Proof.** The primitives . and ′ could be used to generate the operation + as follows :

$(x' . y')' = x + y$. (DeMorgan's Theorem), hence '+' can be defined in terms of . and ′, thus $\{ . , ' \}$ forms the set which is functionally complete (Proposition 3.3).

Similarly the completeness of $\{ + , ' \}$ could be proved.

### 3.5.1. Two Variable Functions

We have till now considered only three operations (three functions of two variables). Actually there exist $2^{2^2} = 16$ functions of two variables. This is so because there are exactly four minterms of two variables and they could be present in 16 various combinations as disscussed below.

Any of the four minterms $x'y'$, $x'y$, $xy'$ and $xy$ could be present or absent in a given switching function. Let us associate four logical parameters A, B, C and D respectively for the above minterms. A parameter assumes a value '1' if a corresponding minterm is present otherwise it is '0'. Thus,

$$f_{ABCD}(x, y) = A.x'y' + Bx'y + Cxy' + Dxy$$

Table 3.3 shows these functions (operations) with their names.

Some of the primitives (from Table 3.3) are commonly encountered in logic design. Hence we shall study them briefly in the following paragraphs.

**XOR Operation**

$$x \oplus y = x'y + xy'$$

$x \oplus y$ is 1 only when $x$ is 1 or $y$ is 1 but not when both $x$ and $y$ are 1.

Truth Table

| $x$ | $y$ | $x \oplus y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Following properties are obvious :

**Properties**

1. Cummutative : $x \oplus y = y \oplus x$
2. Associative : $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
3. If $x \oplus y = z$ then $y \oplus z = x$
   $x \oplus y \oplus z = 0$
4. Distributive : $x.(y \oplus z) = (x.y) \oplus (x.z)$

*i.e.*, AND distributes over $\oplus$.

TABLE 3.3. 16 Functions of two variables

| *Parameters* A | B | C | D | *Symbol* | *Expression* f | *Name of the operation* |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 0 | Inconsistency |
| 0 | 0 | 0 | 1 | $x.y$ | $xy$ | AND |
| 0 | 0 | 1 | 0 | | $xy'$ | Inhibition |
| 0 | 0 | 1 | 1 | | $xy'+xy=x$ | |
| 0 | 1 | 0 | 0 | | $x'y$ | Inhibition |
| 0 | 1 | 0 | 1 | | $y$ | |
| 0 | 1 | 1 | 0 | $x \oplus y$ | $x'y+xy'$ | XOR (Exclusive OR) |
| 0 | 1 | 1 | 1 | $x+y$ | $x+y$ | OR |
| 1 | 0 | 0 | 0 | $x \downarrow y$ | $x'y'$ | NOR (dagger) |
| 1 | 0 | 0 | 1 | $x \odot y$ | $x'y'+xy$ | Equivalence (coincidence) |
| 1 | 0 | 1 | 0 | NOT $y$ | $y'$ | NOT |
| 1 | 0 | 1 | 1 | $y \rightarrow x$ | $y'+x$ | Implication ($y$ implies $x$) |
| 1 | 1 | 0 | 0 | NOT $x$ | $x'$ | NOT |
| 1 | 1 | 0 | 1 | $x \rightarrow y$ | $x'+y$ | Implication ($x$ implies $y$) |
| 1 | 1 | 1 | 0 | $x \uparrow y$ | $x'+y'$ | NAND (Sheffor's stroke) |
| 1 | 1 | 1 | 1 | | 1 | Tautology |

**Equivalence Operation**

$$x \odot y = xy + x'y'$$

The following properties are obvious :

1. $x \odot y = \overline{x \oplus y}$
2. Commutative : $x \odot y = y \odot z$
3. Associative : $x \odot (y \odot z) = (x \odot y) \odot z$
4. Distributive : $x + (y \odot z) = (x+y) \odot (x+z)$
5. If $x \odot y = z$, then $x \odot z = y$
   $x \odot y \odot z = 1$
   $y \odot z = x$.

Truth Table

| $x$ | $y$ | $x \odot y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**NOR/NAND Operations**

NOR : $x \downarrow y = x'y'$

NOR TABLE

| $x$ | $y$ | $x'y'$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**Theorem 3.4.** NOR is functionally complete.

**Proof.** We shall prove the completeness of NOR by proving that ' (NOT) and '+' could be expressed using NOR.

$$x \downarrow y = x'y'$$

$$(x \downarrow y) \downarrow (x \downarrow y) = (x'y') \downarrow (x'y')$$
$$= (x'y' + x'y')'$$
$$= (x+y).(x+y)$$
$$= x+y$$

and $x \downarrow x = \overline{x+x} = x'.x'$
$= x'.$

Thus, $\{+, '\}$ can be generated using NOR operator, since $\{+, '\}$ is functionally complete NOR is a functionally complete primitive.

NAND : $x \uparrow y = x' + y'$

**Theorem 3.5.** NAND is functionally complete.

**Proof.** Similar to that of theorem 3.4.

## 3.6. COMBINATIONAL LOGIC DESIGN

Major application of the theory discussed in the preceding sections is in the design of Digital Logic Networks. These networks fall in two catagories : (1) Combinational and (2) Sequential.

Combinational logic networks are characterised by the output Boolean functions which are dependent on the present inputs. While the sequential network outputs depend not only on the present inputs but also on some past history of the inputs and/or outputs. Usually the past history is stored in some form by storage (or delay) elements. In this section only combinational logic networks are to be studied.

A combinational logic network is composed of various functional modules, interconnected in a specified manner such that the desired output function is implemented. The modules are available as digital integrated logic circuits. These electronic circuits have inputs and outputs as electrical voltages. Usually two distinct voltage levels are assigned to the two logic values '1' and '0'.

**Definition 3.7.** A module is said to implement a Boolean function with positive logic, if the higher voltage level is assigned to logical 1 and lower voltage level is assigned to '0'.

**Definition 3.8.** The module is said to implement a Boolean function with negative logic, if the lower voltage is assigned to logic '1' and higher voltage is assigned to logic '0'.

**Example 3.13.** Consider a device which has input/output characteristics as shown in Table 3.4.

We shall examine what logical function this device represents under negative and positive logic.

TABLE 3.4

| *INPUTS* | | *OUTPUT* |
|---|---|---|
| $x$ *Volts* | $y$ *Volts* | $f$ *Volts* |
| 0 | 0 | 5 |
| 0 | 5 | 0 |
| 5 | 0 | 0 |
| 5 | 5 | 5 |

1. **Negative Logic.** By definition 3.8 we have logical 1 as 0 volts, and logic '0' as 5 volts Substituting these in the Table 3.4, we get a truth Table shown in Table 3.5 (*a*), of the function evaluated by this device under negative logic.

TABLE 3.5 (*a*)

| $x$ | $y$ | $f$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

TABLE 3.5 (*b*)

| $x$ | $y$ | $f$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Looking at the Table 3.5 (*a*), it is easy to see that this device implements a Boolean function given by $f=x \oplus y$.

2. **Positive Logic.** In this case '1' is assigned to higher voltage *i.e.*, 5 volts, and 0 to 0 volts. The Table 3.5 (*b*) shows the truth table formed by substituting these in Table 3.4. This truth table represents a Boolean function $f=x \odot y$

Throughout this book we shall assume that the logic modules work on positive logical basis, so that there is no confusion regarding the functions they implement.

Commonly used function modules (called gates) are listed with their network symbols in Table 3.6. Using these symbols we can sketch logic networks implementing Boolean functions.

**Example 3.14.** Design a combinational logic networker to implement the function $f=w'x+w.y+z'x'+y'x$.

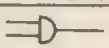
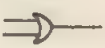
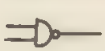
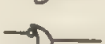
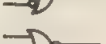
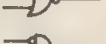
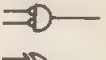
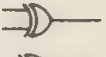
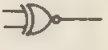
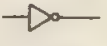
Assuming that we have NOT, AND, and OR modules with any number of inputs we get the logic network of Fig. 3.6 for this function. See Fig. 3 6 carefully.

### 3.6.1. Logic Design Example

In this subsection complete engineering design of logic circuit is presented for simple hypothetical signalling system as described below.

**Example 3.15.** There are four parallel railway tracks at a place. It is desired to know whether at this place three or more trains pass at any given time. Design an

TABLE 3.6 : 2-Input Logic Gates and their network symbols

| Name | Logic symbols in the network | Boolean function implemented |
|---|---|---|
| AND |  | f(x, y) = x y |
| OR |  | f(x, y) = x + y |
| NAND |  | f(x, y) = $\overline{x \cdot y}$ = x' + y' |
| NAND |  | f(x, y) = x' + y' |
| NOR |  | f(x, y) = x'. y' |
| NOR |  | f(x, y) = x'. y' |
| XOR |  | f(x, y) = x'y + x y' = x ⊕ y |
| EQUI |  | f(x, y) = xy + x'y' = x ⊙ y |
| NOT |  | f(x) = x' |
| NOT |  | f(x) = x' |

Note : x and y are input variables.

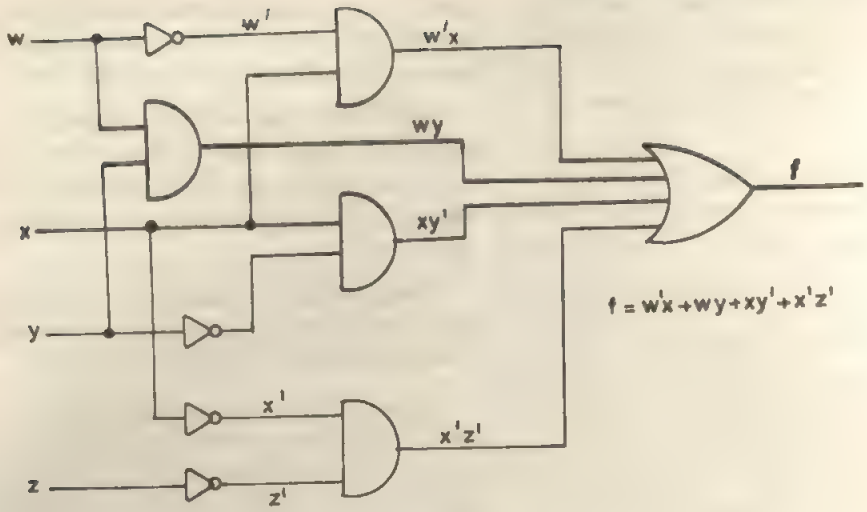



FIG.3·6 EXAMPLE LOGIC NETWORK

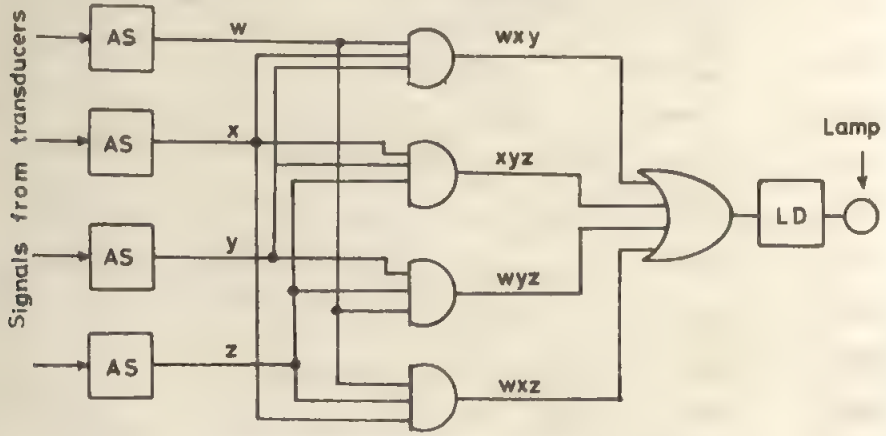
automatic signalling system, to signal the above condition.

We assign variable $w$, $x$, $y$ and $z$ to denote the presence of trains on the four tracks respectively. A pressure transducer can be used to pick up the signal for the presence of a train on a track, at a required point. Voltages picked up could be amplified and shaped into two levels so as to represent '1' and '0' for two conditions. (Presence or absence of a train).

A truth table can now be constructed as shown in Table 3.7 to indicate the desired condition (3 or more1's on the input variables $w$, $x$, $y$ and $z$) and the function $f$ is entered in the Karnaugh Map. The simplified expression for $f$ is given by :

$$f = xyz + wxy + wxz + wyz$$

The logic network implementing $f$ and a complete signalling system is shown in Fig. 3.7.




AS : Amplifier shaper, LD: Lamp driver

FIG.3·7 LOGIC DESIGN EXAMPLE

TABLE 3.7

| $w$ | $x$ | $y$ | $z$ | $f$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

| yz \ wx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 0 |
| 11 | 0 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 0 |

### 3.6.2. Logic Design With NAND/NOR

Any switching function can be expressed using either NAND or NOR operations. Thus it is possible to implement logic networks using only NAND or NOR gates.

Designing with NAND/NOR poses some difficulties usually not encountered in designing with AND, OR and NOT logic. This is primarily due to the fact that switching functions could be easily expressed using AND, OR and NOT operations. The main difficulty with NAND/NOR design lies in the fact that in order to obtain NAND/NOR realisations, corresponding algebraic expression must be 'factored' in such a way that NAND/NOR will be the only operation in the expression. This step is usually quite complicated because it requires a large number of applications of DeMorgan's theorem. Example 3.16 illustrates this procedure.

**Example 3.16 :** Design a two-input XOR gate using only (*a*) NANDS

(*b*) NORS.

The function to be implemented is

$$f = x'y + xy'$$

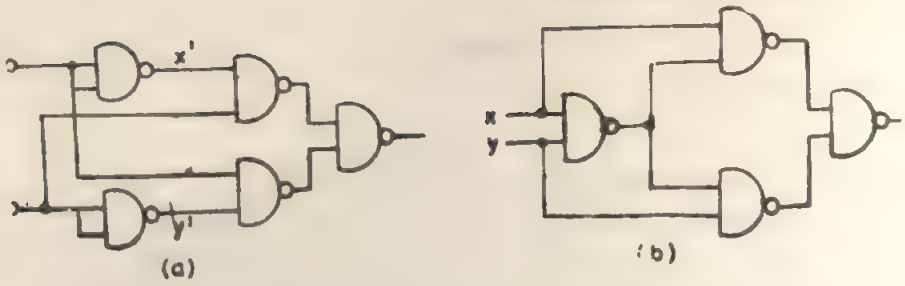



FIG.3·8 XOR IMPLEMENTATIONS USING ONLY NAND GATES

**(b) NAND Implementation :**

$$f=f''=\overline{\overline{x'y+y'x}}$$
$$=\overline{(x+y')\,.\,(x'+y)}$$
$$=\overline{\overline{x+y'}\;.\;\overline{(x'+y)}}$$
$$=\overline{\overline{x'\,.\,y}\;.\;\overline{y'\,.\,x}}$$
$$=\overline{\overline{x.x.y}\;.\;\overline{y.y.x}}$$
$$=((x\uparrow x)\uparrow(y))\uparrow((y\uparrow y)\uparrow x)))$$

where '↑' denotes the NAND operation. The above expression uses only ↑ operator and hence could be implemented as shown in Fig. 3.8 (*a*).

Alternate expression and implementation is obtained as follows :

$$f=xy'+x'y$$
$$=\overline{(x'+y)(y'+x)}$$
$$=\overline{(x'+xy)\;.\;(y'+xy)}$$
$$=\overline{\overline{xy'.x}\;.\;\overline{x.y'.y}}$$
$$=((x\uparrow y)\uparrow x)\uparrow((x\uparrow y)\uparrow y$$

The alternate implementation is shown in Fig. 3.8. (*b*). Note that here same logic is implemented using only four NANDS (this thus is less costly).

**(b) NOR Implementation** : Exercise

The theoretical and applications oriented treatment have been presented in this chapter on the Boolean Algebra and logic design. The examples are given to help readers in better understanding of the material presented. Next section discusses some commonly used combinational circuits.

## 3.7. SOME COMMONLY USED COMBINATIONAL CIRCUITS

Decoders, encoders, multiplexors and demultiplexors are some of the combinational circuits which find applications in many digital systems. This section is therefore devoted to the discussion of these circuits.

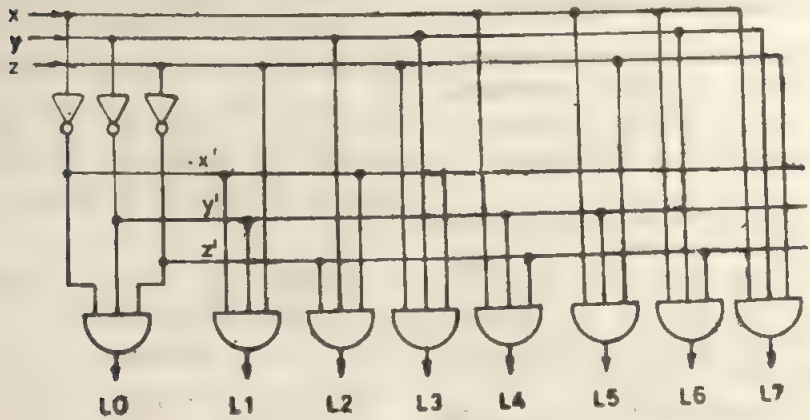



FIG. 3·9 3 TO 8 DECODER

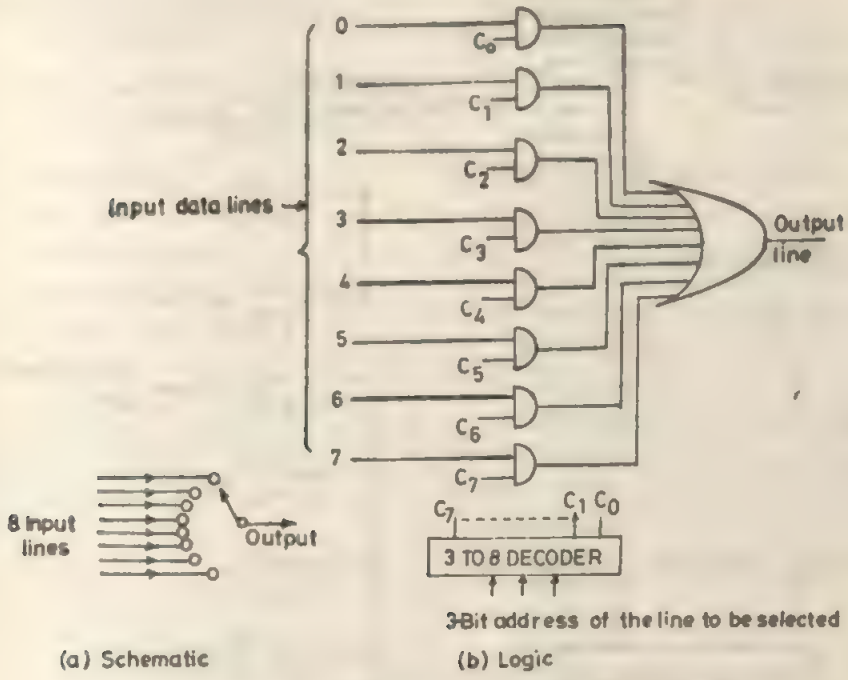



FIG.3·10 8 TO 1 MULTIPLEXOR (DATA SELECTOR)

### 3.7.1. Decoders

A decoder is a combinational circuit whose function is to 'decode' combinations of input signals. A decoder with $k$ input and $m$ ($m \leq 2^k$) output lines, where $k$, $m$ are integers, produces an active (say logical '1') signal on one of the $m$ output lines, while other output lines are inactive (logical '0'). A decoder with $k$ input lines and $m$ output lines is referred to as $k$ to $m$ decoder. Consider a 3 to 8 decoder shown in Fig. 3.9. In this decoder, the output line $L_i$ takes value '1' if the input combination represents the number '$i$'. Thus, line $L_0$ is '1' if $xyz$=000, $L_1$=1 if $xyz$=001, and so on. In Fig. 3.9, the eight output lines are the outputs of eight AND gates, each producing a true signal for a particular input combination.

### 3.7.2. Multiplexor (Data Selector)

The function of a multiplexor is to select data from one of the many data lines. A multiplexor with $m$ input lines out of which one is to be selected is called a $m$ to 1 multiplexor. Usually $m$ is a power of 2. The logic for 8 to 1 multiplexor is shown in Fig. 3.10. In this logic circuit, each data line is gated with a control signal ($C_i$ for $I^{th}$ line) by an AND gate. The multiplexor has 3 'address lines' which indicate a line to be selected. The address lines are decoded by 3 to 8 decoder which produces 8 control signals $C_0$, $C_1$ ..$C_7$. Since only one of $C_i$ is true, the data from $I^{th}$ line appears at one of the output AND gate, and other AND gates produce '0' at their output, which passes to the output line.

The multiplexor discussed selects one of the 8 data lines To implement selection of say one number out 8 numbers, each of length, say, 16 bits, we shall simply put 16, 8 to 1 multiplexors of Fig. 3.10, all of which shall be driven by common address lines.

### 3.7.3. Demultiplexor (Data Distributor)

The use of demultiplexor is to distribute

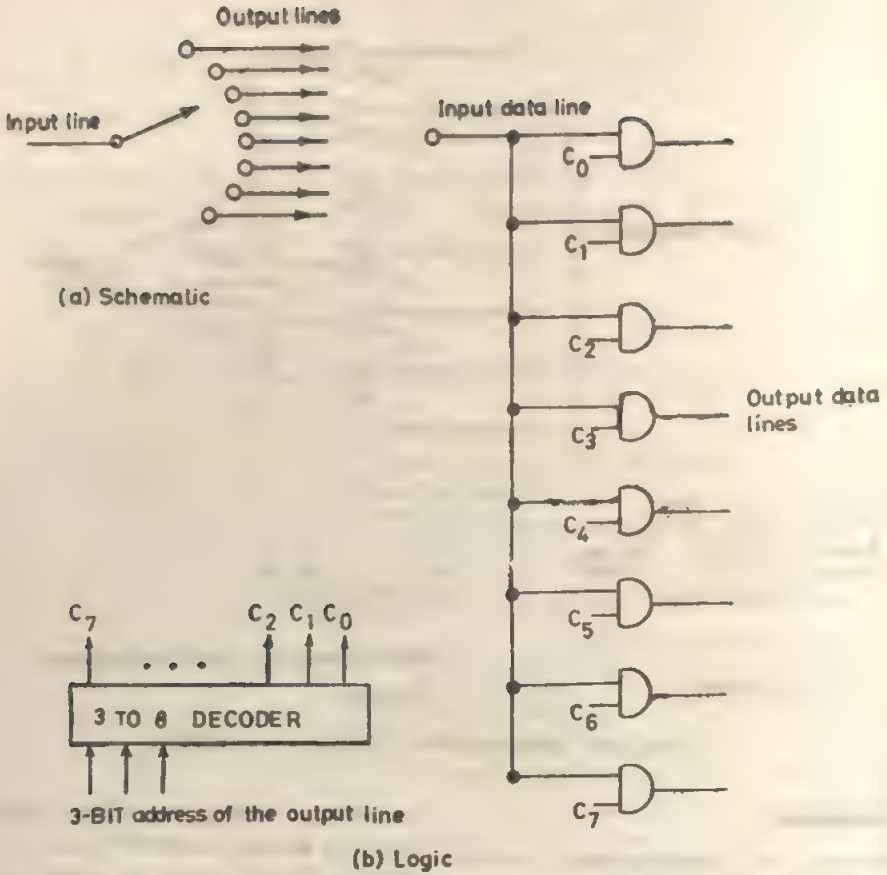



FIG 3·11 1 TO 8 DEMULTIPLEXOR (DATA DISTRIBUTOR)

data coming on an input line to one of the selected $m$ lines. The schematic of demultiplexor is shown in Fig. 3.11 (*a*). This diagram is same as Fig. 3.10 (*a*) except that signals flow in the opposite direction. Since logic circuits are unidirectional we cannot use the circuit of Fig. 3.10 (*b*) to realise the demultiplexor. The Fig. 3.11 (*b*) shows the logic circuit of a 1 to 8 demultiplexor. In this logic circuit input line data is passed to one of the required O/P lines by opening the gate of that line. The gate open signals are again obtained from a 3 to 8 decoder as shown in Fig. 3.11 (*b*).

## EXERCISES

1. Simplify the following expressions using theorems and properties of Boolean Algebra :

(*a*) $x+y+x'y'z$

(*b*) $(x+x'y'z)+(x+x'y'z)\,.\,(x'+xyz')$

(*c*) $x'y'+w'x'y'z+xy'$

(*d*) $x_1+x_1'x_2+x_1'x_2'x_3+x_1'x_2'x_3'x_4+\ldots$

(*e*) $x'y'+yz+w'x'z$

(*f*) $wx+xy+wz+y'z'$

2. Find the complement for each of the following and then simplify :

(*a*) $x(y+z)'\,.\,(x'+y'+z)$

(*b*) $wx\,(y(z+x)+w'x')+(wx'+zy)$

(*c*) $(A+B)\,.\,(C+D)\,.\,(B+A)\,.\,(A+B).\,(C+D))$

3. Given $z=xy'+x'y$ show that
$$y=z'x+x'z$$

4. Find the values of the variables $w,x,y$ and $z$ by solving the following simultaneous equations.
$$w'+wx=0$$
$$wx=wy$$
$$wx+wy'+yz=y'z$$

5. Prove that if $A'B+CD'=0$ then
$AB+C'(A'+D')=AB+BD+B'D'$
$+A'C'D$.

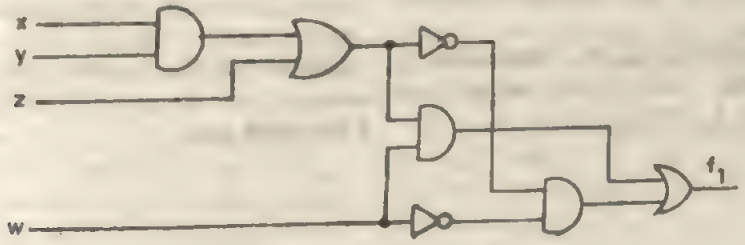
6. Let the connective (operation)* is defined as $A*B=AB+A'B'$ and let $C=A*B$. Find which of the following are valid :
(*a*) $A=B*C$
(*b*) $B=A*C$
(*c*) $A*B*C=1$

7. Determine the canonical sum of products representations for the following functions and simplify them using Karnaugh maps.
(*a*) $f(x,y,z)=z'+(x'+y')(x+y')$
(*b*) $f(x, y, z)=x+yz+yzx'$

8. Fill up the truth tables for the functions below and write down the canonical product of sums expression for them and simplify them using Karnaugh maps.
(*a*) $f(A, B, C)=AB+AC$
(*b*) $f(x, y, z)=x+y'z$

9. Determine the minimal sum of products expression for
(*a*) $f=\Sigma(0, 2, 4, 9, 12, 15)+\Sigma(1, 5, 7, 10)$
(*b*) $f=\Sigma(1, 5, 6, 7, 8, 9, 10, 14, 15)$

10. Find the maxterms of the functions in



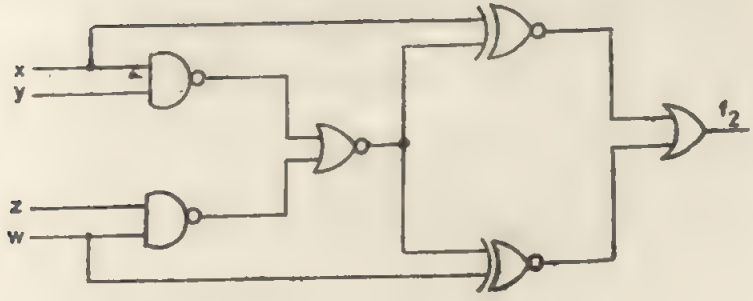




FIG. 3-12 LOGIC NETWORKS

(9) and write down the simplified product of sums expression.

11. A safe has five locks, A, B, C, D and E. All of these must be unlocked so as to open the safe. The keys to these locks are distributed among the five persons in the following manner :

Mr. Pitke has keys for looks A and C
Ms. Aruna has keys for locks A and D
Mr. Rajiv has keys for locks B and D
Mr. Dicken has keys for locks C and E
Ms. Chitra has keys for locks A and E.

(*a*) Find all the possible combinations of persons that must be present to open the lock.

(*b*) Find the essential persons who must always be present.

12. Let $f=\Sigma(1, 2, 6, 13)$ and $f_1=\Sigma(0, 1, 2, 3, 5, 6, 8, 10, 11, 13)$. Find $f_2$ such that $f=f_1.f_2$. Is $f_2$ unique ? If not indicate all possibilities.

13. Let $f=\Sigma(0, 11, 12, 15)$. Determine $f_2$ and $f_3$ such that $f=f_3+f_1.f_2$, where $f_1=\Sigma(0, 1, 2, 3, 5, 6, 8, 10, 11, 13)$.

14. Express $f_1$ and $f_2$ as functions of $w$, $xy$ and $z$ for network of Fig. 3.12.

15. Give the equivalent implementation for $f_1, f_2$ of (14) using

(*a*) NANDS
(*b*) NORS

16. A three input gate called 'EXPLODE' whose table is shown below is manufactured by some unfortunate company.

| x \ yz | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |

Experimental evidence shows that input combinations 101 and 010 cause the gate to explode. Your task is to determine whether this explode gate is completely useless or can be externally modified so that it may be used for implementing any switching function. If your answer is yes then give the external logic circuit and its connections to EXPLODE gate.

□

# 4

# DIGITAL SYSTEMS BUILDING BLOCKS

IN this chapter we shall study logic circuits which permit the storage and elementary manipulation of digital information. A class of circuits called sequential circuits are usually built using digital storage devices. This chapter is devoted to the discussion of logic circuits which are commonly used in building digital systems.

## 4.1. INTRODUCTION TO SEQUENTIAL LOGIC CIRCUITS

Combinational logic circuits studied in chapter 3 have the property that the outputs of these circuits depend only on the applied input combination and are completely independent of the past inputs or outputs. Sequential logic circuits are different from their combinational counterparts. Output of a sequential circuit not only depends on the present input combination but also on some past input and/or outputs. A sequential logic can be modelled by a block diagram shown in Fig. 4.1.

In Fig. 4.1, a combinational logic circuit has some inputs derived from the memory elements (delays) which remember some past history of the network. The past history may be the outputs or inputs at some earlier time instances.

As seen in Fig. 4.1, a sequential logic has feedback paths. In this chapter we shall study a basic component of a sequential logic called flip-flop (F/F) and use it to construct some simple oftenly used sequential logic circuit blocks *i.e.*, registers, counters etc.

## 4.2. FLIP-FLOPS

A flip-flop is constructed using logic gates

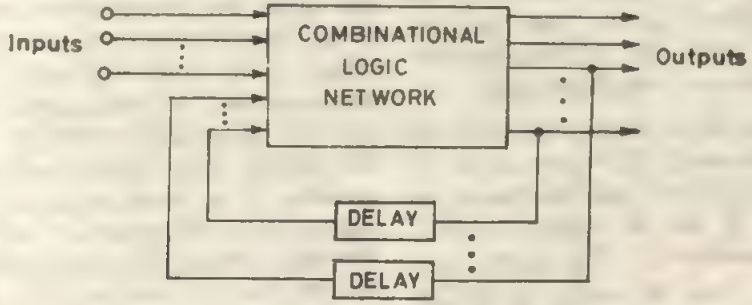



FIG. 4·1 SEQUENTIAL LOGIC CIRCUIT MODEL

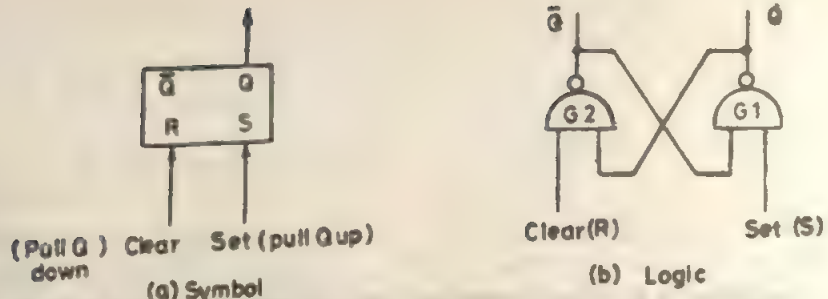



FIG. 4·2 R-S ASYNCHRONOUS FLIP-FLOP

to store a bit of information. The logic symbol of a flip-flop is shown in Fig. 4.2. Variety of flip-flop types could be constructed by defining various input logic functions.

### 4.2.1. Asynchronous Flip-Flop

The simple flip-flop (F/F), shown in Fig. 4.2 is called an asynchronous R-S (Reset-Set) flip-flop.

The basic function of a flip-flop, as said earlier, is to store a given bit of information. Thus one may like to put a '0' or '1' in the flip-flop (the logic line Q is made 0 or 1). The input R is usually used to put a '0' in F/F (Reset) while the input S is used to put a 1' (set). The operation of putting a '0' or '1' in F/F of Fig. 4.2 is analogous to raising a flag on the flag post by a thread if two ends of thread are available one to lower and one to raise the flag. If there exists some holding mechanism, the flag can be kept in upper or lower position even after removing the forces on the thread ends. This principle is also true for a F/F. Fig. 4.2 (*b*) shows how this could be achieved using logic gates.

The operation of the circuit of Fig. 4.2 (*b*) is as follows : Normally R, S inputs are in '1' state (when we do not want to change the F/F state). If Q was its output, we shall show that it remains at Q. The gate 2 has inputs Q and R, thus its output is $1.\overline{Q}=\overline{Q}$. The gate 1 has the inputs $\overline{Q}$ and S, therefore, the output of gate 1 is $1.\overline{Q}=Q$. Thus if Q was 0, it is maintained at '0'. On the other hand if Q was '1' it is maintained at a '1' level. Hence this circuit remembers a bit of information. Thus a F/F is the basic memory cell (storing a bit of information) in digital logic circuits.

If R is made '0' and S is kept at 1, then output of gate 2 is $0.Q=1$ and output of gate 1 is $1.1=0$, giving Q a value '0'. Now if the line R is taken to 1 from its previous zero value, Q still maintains a logic '0'. Similarly if the line S is made momentarily zero while R is at '1', Q assumes a value '1'. Thus the inputs R and S can make Q to store a '0' or '1'. This flip-flop is called an asynchronous flip-flop. Since the output of the gate 2 is at a complementry state to that Q, it is labelled as Q. A F/F usually has both Q and Q as available outputs.

### 4.2.2. Synchronous Flip-Flops

Though asynchronous F/Fs are nevertheless useful, many applications require change of the state of a F/F by a special pulse signal called clock, which represents the known time instances. A clock signal is shown in Fig. 4.3. In synchronous F/Fs, a change in the flip-flop state occurs only due to the presence of a clock pulse. A variety of input logic could be defined for changing a F/F state. This leads to the different types of F/Fs. We shall study these in the following subsections.

Synchronous flip-flops change their state only when a clock pulse is applied. A table

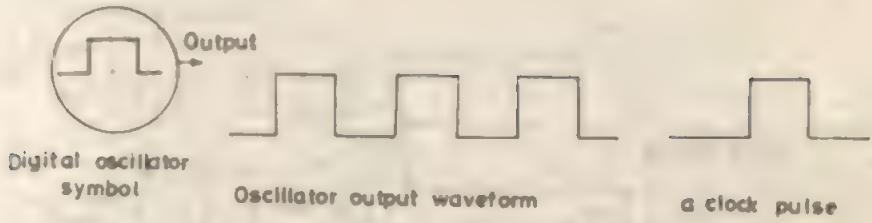



FIG 4·3 CLOCK

called transition table or next state table is usually used to describe the behaviour of synchronous F/Fs. The transition table has a number of rows which equals the possible number of combination of the inputs (excluding clock) and for each combination, an output is specified. This output (next state) is assumed to be present after a clock pulse has been given to the clock terminal of a F/F.

**Clocked R-S Flip-Flop**

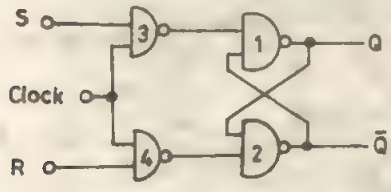
The logic circuit of this F/F is shown in Fig. 4.4 (*a*) and its transition table is shown in Fig. 4.4 (*b*). The F/F consists of a basic F/F cell discussed earlier and a clock gating logic.

The R-S clocked F/F works as follows : A clock signal is gated with S-R inputs on the input gating NAND gates, and depending upon the values of R, S the clock (with inversion) appears at gate 3 or 4. If R=0 and S=1, the inverted clock appears at gate 3. This provides a momentary '0' at set input of the basic F/F cell, composed of gates 1 and 2. As per the earlier discussion Q will be set to '1' and consequently $\bar{Q}$ will take a value '0'. Thus a '1' is stored in the F/F. On the other hand if R=1 and S=0 gate 2 input will get a momentary '0' and its output ($\bar{Q}$) will become '1', thus storing a '0' in Q. If both R and S are at '0' and clock pulse is given Q will not be affected since the clock does not reach any of the gates 1 and 2, thus leaving Q in its old state. On the other hand, if R=S=1, then the clock reaches both the inputs of gates 1 and 2 making Q and $\bar{Q}$ both logical one, till the clock is present. After the clock pulse goes off, it is not possible to predict the next state, hence a '*' is entered in the transition table for this combination.

**J-K Flip-Flops**

J-K F/Fs are characterised by the transition table and logic shown in Fig. 4.5.

The J-K and R-S F/F of Figure 4.5 and

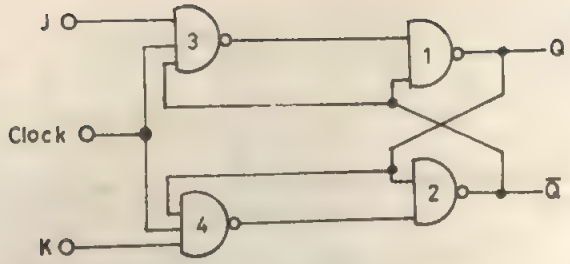



(a) Logic diagram

| S | R | $Q_{next}$ |
|---|---|---|
| 0 | 0 | $Q_{old}$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | * |

(b) Transition table

FIG. 4·4 SYNCHRONOUS R-S FLIP-FLOP




(a) Logic diagram

| J | K | $Q_{next}$ |
|---|---|---|
| 0 | 0 | $Q_{old}$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q_{old}}$ |

(b) Transition table

FIG. 4·5 J-K FLIP-FLOP

Figure 4.4 respectively have same number of logic gates and connections with the difference that the input gates have Q and $\bar{Q}$ as additional inputs. This is made to define a fixed state of $\bar{Q}$old for 1, 1 combination of J, K inputs, unlike an undefined state of R-S F/F for this combination. The circuit of figure 4.5 functions exactly like R-S clocked F/F for all input J, K combinations except when J=K=1. In the case of 1, 1 combination, due to the feedback connections from Q and $\bar{Q}$, the clock will make Q high if Q was low earlier ($\bar{Q}$=1 hence clock passes through the gate 3). On the other hand if Q (old)=1, $\bar{Q}$ is made high due to the passing of the clock through the gate 4, therefore, the clock actually complements the F/F state for the 1, 1 combination of J and K inputs.

The J-K F/F discussed works properly for J=K=1 combination if the delay of the F/F nearly equals the width of the clock pulse. However, if the pulse width is larger than the F/F delay, the action of clock will be continuously present and, the F/F state will undergo numerous complementations (till the clock is high). Depending upon the pulse width, the J-K F/F would have undergone a number of complementations before it has settled (after clock goes low). The evenness or oddness of this number will finally decide what state Q will have. This problem is called racing and has to be avoided. The most common approach to solve this problem is to use two F/F cells as follows.

**Master-Slave J-K F/F**

The logic circuit for a master slave J-K F/F is shown in Fig. 4.6. There are two F/Fs FF1 and FF2 clocked by complementary clocks. The clock signal CK is normally at a logic '0'. This disables the master F/F (gates 3 and 4) from the input logic (gates 1

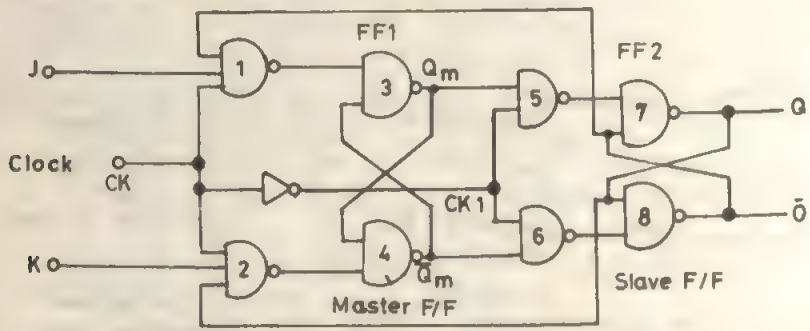



FIG. 4·6 MASTER SLAVE J-K F/F LOGIC

and 2), thus the master F/F retains its previous value. At this time the clock CK1 ($\overline{CK}$) is at a logical '1', producing $Q_m$ and $\overline{Q_m}$ on the gates 5 and 6 respectively. If $Q_m$ is 0, the gate 8 output becomes 1 making gate 7 output 0, thus $Q_m$ and Q are '0'. On the other hand if $Q_m=1$, the gate 8 output becomes 0 and gate 7 output becomes '1'. Thus the line Q (output of gate 7) follows the line $Q_m$. This is the reason why the FF1 is called a master and FF2 is called a slave (*i.e.*, slave F/F follows the state of master (FF1) when CK=0). When the clock CK becomes '1' the clock CK1 goes low and the outputs of gates 5 and 6 are '1'. Therefore, FF2 is logically disconnected from FF1. At this time the old value of the master is replaced as per J-K combination (the slave, since disconnected, retains the old value). Now if the clock CK is made low, slave follows the master and the changed state of master is followed by FF2. This arrangement even works for J=K=1 combination, since the feedback is taken from the slave F/F, whose new state is entered only after the clock pulse has gone.

**Edge Triggered Flip-Flops**

A number of F/F types could be built which characterises the state transitions at the leading or trailing edge of the clock pulse rather than on the clock itself. This implies that only the clock transition interval is of importance for the triggering (*i.e.*, changing of a state) These F/F's could be constructed using a digital differentiator circuits discussed below.

**Digital Differentiator.** Often, it is necessary to produce sharp pulses at the leading and/or trailing edges of a given clock pulse as is done by a conventional RC differentiator. For example, an output of the RC differentiator will look like as the one shown in Fig 4.7 (*a*). We are interested in the pulses in the voltage range of a logic family, say 0 to 5 volts. A negative voltage pulse of the RC differentiator has to be inverted in polarity if it is required to be

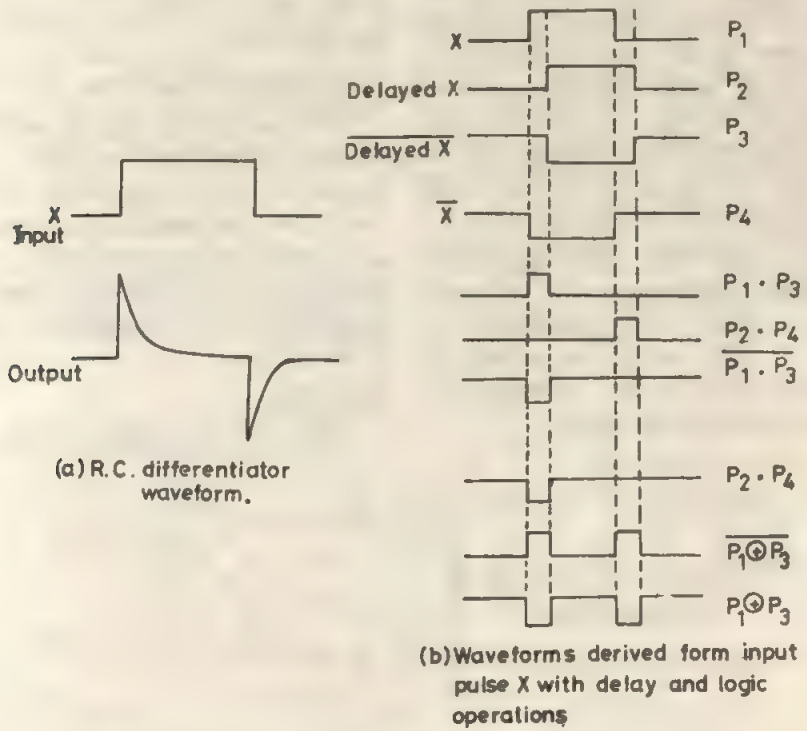



FIG. 4-7 DIGITAL DIFFERENTIATION PRINCIPLE

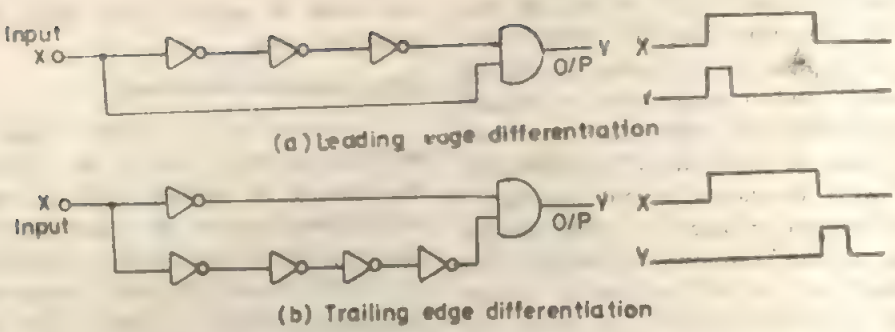



FIG 4·8 DIGITAL DIFFERENTIATORS

used. The differentiation can well be accomplished using logic gates in an inexpensive way as follows. Since these circuits are carrying out the equivalent of differentiation, we shall call them as digital differentiators.

Consider the waveforms $P_2$, $P_3$, $P_3$ and $P_4$ shown in Fig. 4.7 (*b*). These waveforms are obtained from an input pulse X by delay and/or inversions. They could be suitably combined using AND/OR logic gates to produce the required small duration pulses. For example, a sharp pulse at the leading edge of a given pulse can be obtained by 'ANDing' an input pulse with the inverted delayed input as shown in Fig. 4.8 (*a*). Fig. 4.8 (*b*) shows the logic circuit for obtaining a sharp trailing edge pulse. In such logic circuits, the delay required could be obtained by using delays in the logic gates themselves.

Edge triggered F/Fs can now be constructed easily using the principles of differentiators. A F/F is constructed with an internal clock which is the differentiated output of the external clock signal.

Depending upon whether a leading or a trailing edge differentiator is used, the F/F can be constructed to trigger at the leading or the trailing edge of the external clock.

**Edge Triggered JK Flip-Flop**

Consider a F/F shown in Fig. 4.5. This J-K F/F will work properly if the clock width is made nearly same as FF delay. F/F delay is equivalent to the delay of 2 logic gates. Fig. 4.9 shows an edge triggered JK F/F triggering at the trailing edge (the external clock passes through the trailing edge differentiator).

**Other Types of Flip-Flops**

Two more important flip-flops which are usually encountered in logic circuits are Toggle F/Fs (TF/Fs) and Data F/Fs (DF/Fs). The former are usually required in constructing counters while latter are used in registers. Nevertheless, any flip-flop type could be modified externally to behave as required for its input logic.

**T-Flip-Flop**

The transition table and the logic circuit

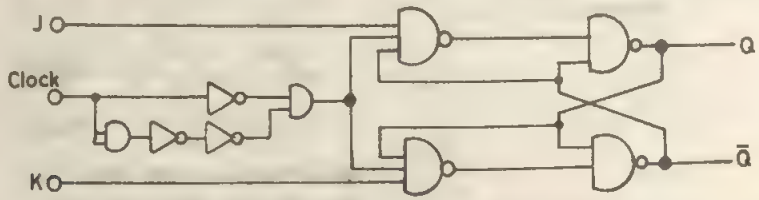



FIG. 4·9 EDGE TRIGGERED J-K FLIP-FLOP.

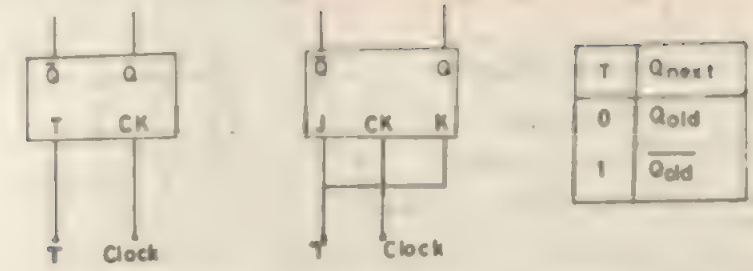



FIG. 4.10 T (TOGGLE) FLIP-FLOP

for this F/F is shown in Fig. 4.10. The F/F output gets complemented for every clock pulse in T F/F if input T=1, otherwise nothing happens to Q. This is implemented by connecting J-K inputs together and calling this connection as T input. Since J=K (due to connection), for J=K=0, *i.e.*, T=0, $Q_n = Q_{old}$ and for J=K=1, $Q_n = \bar{Q}_{old}$ as per the transition table.

**D Flip Flop**

This flip-flop has D as a data input. A clock pulse enters the value on the data line into the F/F. The circuit symbol, transition table and logic circuit using R-S (J-K F/F also can be used) flip-flop is shown in Fig. 4.11.

**Converting One F/F Into Another**

The last two F/Fs studied each use a J K or R-S F/F internally. It is interesting to know whether we can construct a J-K or say, D F/F using T F/F or any other flip-flod available by addition of extra logic. In this section we shall show and study a procedure by which we can always convert a given edge triggered or master slave F/F into any required F/F by adding external logic. The seemingly difficult task of converting a D F/F into J-K F/F will be illustrated now. The logic diagram shown in Fig. 4.12, in fact, does the required job. This can be verified by computing the transition table for J, K using the circuit of Fig. 4.12.

The procedure is very simple. The transition tables of both the F/Fs are known. It terms of these tables, a conversion circuit has to map the transition table of a given F/F into that of a required F/F. The table shown in Fig 4.12 has 3 input variables and one output variable D. This table gives the switching function defining D with J, K and Q acting as inputs.

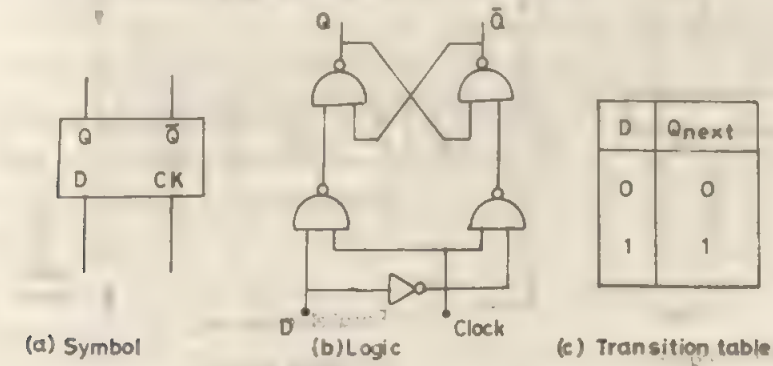
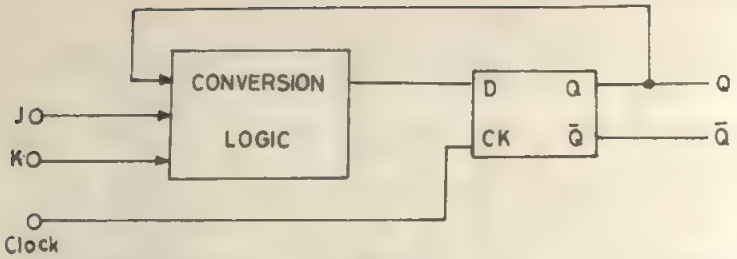


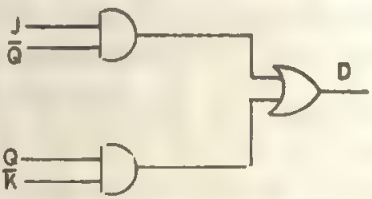

FIG. 4.11. D FLIP-FLOP




(a) D TO J-K FLIP-FLOP CONVERSION LOGIC SCHEMATIC

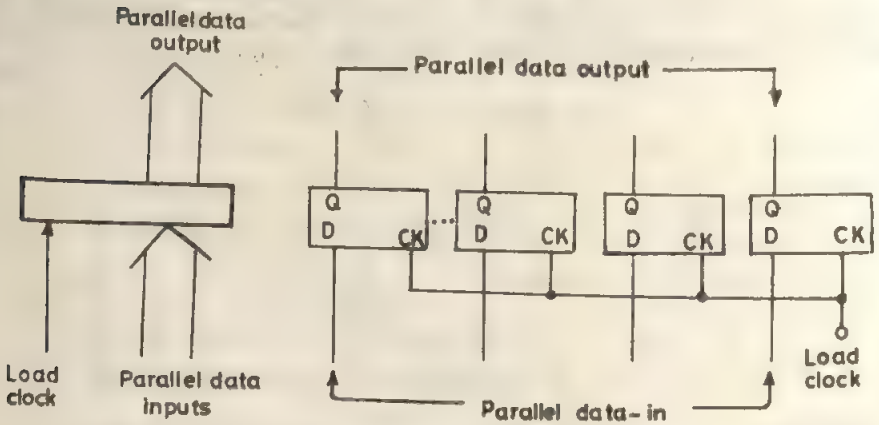
| Input | | | Output |
|---|---|---|---|
| J | K | Q | D |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

(b) Switching function of conversion logic




(c) Conversion logic

FIG. 4-12 D TO J-K FLIP-FLOP CONVERSION




(a) Block diagram

(b) Logic circuit

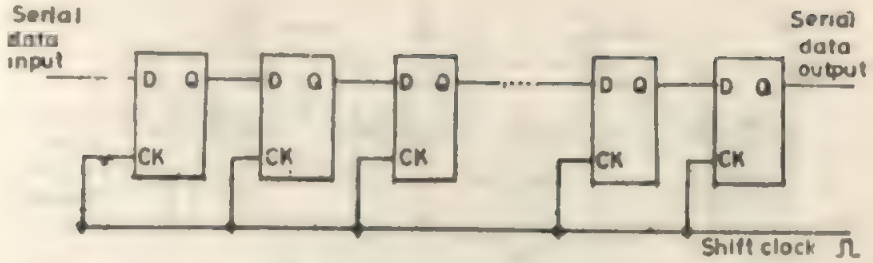



FIG. 4-14 SHIFT REGISTER

## 4.3. REGISTERS

Registers are used to store the binary or binary coded numbers. To store an n-bit number we require an n-bit register. Such a register can be constructed from n flip-flops as shown in Fig. 4.13. All the flip-flops in the register have a common clock so that n-bit data could be entered simultaneously. This operation of entering data into the register is called loading. In Fig. 4.13 (*b*) a register is constructed from D F/Fs.

This register is called 'parallel-in, parallel-out' because of the fact that data could be entered and observed parallelly.

**Shift Registers.** A shift register is made of F/Fs connected as shown in Fig. 4.14. Every clock pulse shifts the contents of the register by a fixed number of bits. The shift register shown in Fig. 4.14 shifts the data to the right by 1 bit. The vacated position is filled by the data on the serial-in line. The data on the serial out line is lost after every clock pulse.

Since the output of a flip-flop is connected to the D input of the right neighbour, every pulse shifts the data in the register by 1 bit right.

In general one may have a shift register with various shifting capabilities. A shift by 1 bit left or right, 2 bit shift etc., may be jointly made available in a shift register by additional logic. There are several types of shift registers available in the Integrated Circuit form.

## 4.4. COUNTERS

Counting is a primitive operation ($x \leftarrow x+1$ (count up) and $x \leftarrow x-1$ (count down). Any arithmetic operation (*i.e.*, add, sub, multiply, divide) can be achieved using counting. In digital systems, counters are devices composed of logic circuits in which the numbers are counted (Inc/Dec by 1). The numbers can be in various codes ; *i.e.*, pure binary, gray code, BCD etc. In this section we shall study binary and BCD (decade) counters. There are two types of counters.

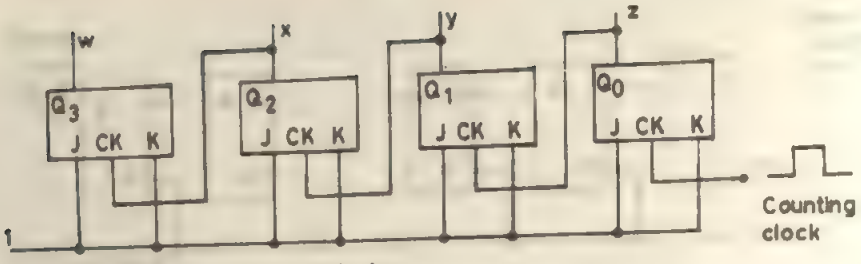
- Asynchronous counters
- Synchronous counters.

In asynchronous counters, the states of all the F/Fs do not change at a precisely given instant although a clock signal is used. Contrary to this, in synchronous counters, all the F/Fs make a transition at the common clock time.

### 4.4.1. Asynchronous Binary (Ripple) Counters

We shall illustrate the principle involved in these counters with the help of 4-bit up counter. The 4-bit binary counter has 4 F/Fs connected in the manner such that the joint state transitions of all the 4 F/Fs are as per the Transition Table shown in Fig. 4.15 for every clock pulse. As can be noted, the value of the 4-bit number in the counter register increases by one for every clock pulse.

Construction of the up counter is equivalent to achieving the state transitions of




(a)

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Q_0$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| $Q_1$ | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $Q_2$ | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $Q_3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(b)

FIG. 4·15 4-BIT ASYNCHRONOUS UP COUNTER

$Q_3$, $Q_2$, $Q_1$ and $Q_0$ F/Fs as per the table of Fig. 4.15 (*b*).

Let us see how we could construct a counter for the said transition table. It can be seen that the bit $Q_0$ (LISB) F/F makes transition for every clock pulse given to the counter, and the $Q_1$ F/F makes a transition when the $Q_0$ F/F makes a 1 to 0 transition. In fact, it can be observed that a F/F in bit position 'i' makes a transition (its state is complemented) if the F/F in bit position i-1 makes a 1 to 0 transition. Therefore, if we use the F/Fs which are triggered at the trailing edge of the clock, Q output of (i-1)th F/F could be connected to the clock input of the ith F/F. This is what is shown in Fig. 4.15 (*a*). The 4 F/Fs have the outputs as shown in waveforms in Fig. 4.16. It may be noted from Fig. 4.16 that the frequency of the waveform at the line $Q_0$ is half of the input clock frequency. Similarly each F/F in the higher order has frequency scaled by half to the frequency of its lower order neighbour. Thus each flip-flop output in the counter represents the counting of pulses by a power of 2.

A four bit down counter decreases the binary number in the counter register by 1

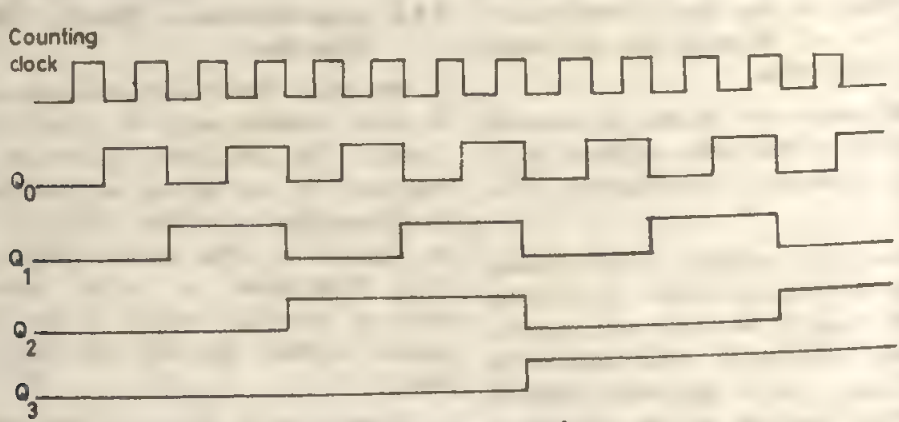



FIG . 4·16 COUNTER FLIP-FLOPS' WAVEFORMS

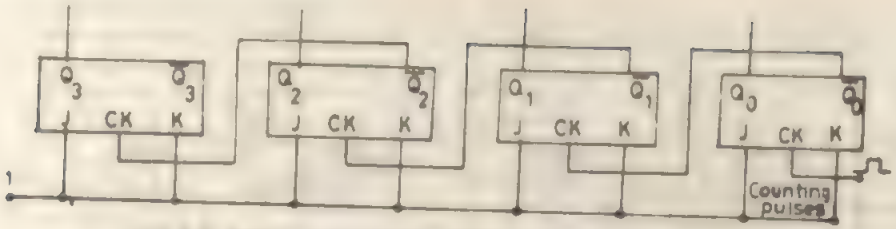



FIG 4-17 4-BIT ASYNCHRONOUS DOWN COUNTER

for every input clock pulse. In other words, the state transitions shown in the table are scanned in the right to left manner. It can be easily seen that such a counter can be constructed using the same logic diagram (Fig. 4.15 (*a*) by using F/F triggering at the leading edge. However, if we use F/F triggering at the trailing edge, we shall have to use the output $\overline{Q}$ of each F/F as a clock for the next neighbouring F/F. This is shown in Fig. 4.17.

### 4.4.2. Synchronous Binary Coudters

Unlike in the asynchronous counters, synchronous counters have the same clock signal for each of the flip-flops. This makes the state transitions of all the flip-flops to occur at the same time instant.

The synchronous binary up counter can be constructed again by noting the relations between a current states and the next state. From the Fig. 4.15 (*b*) it can be noted that a F/F makes a transition if all its previous F/Fs are all in '1' state. For example, the state of the F/F X gets complemented if Y.Z=1, *i.e.*,

(*i*) W X Y Z

0 0 | 1 1 Y=Z=1

0 1 | 0 0

X getting complemented.

(*ii*) 0 1 | 1 1 Y=Z=1

1 0 | 0 0

X getting complemented.

This rule is true for any bit position, while the least significant bit gets complemented for each clock pulse. The logic circuit of a synchronous 4-bit up counter is shown in Fig. 4.18. In a 4-bit down counter a F/F gets complemented if the previous F/Fs are all at logical '0's. Thus in the logic diagram of Fig. 4.18. $\overline{Q}$ should take part in the connections shown for Q if one wants to construct a down counter.

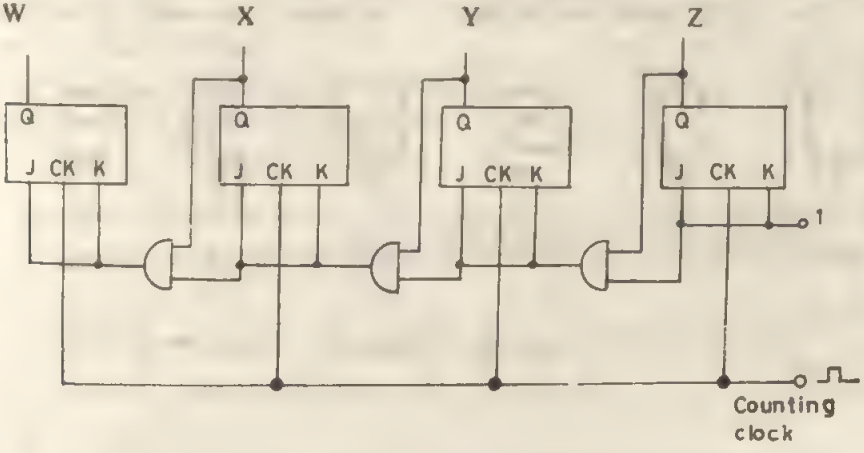



FIG. 4-18 4-BIT SYNCHRONOUS UP COUNTER

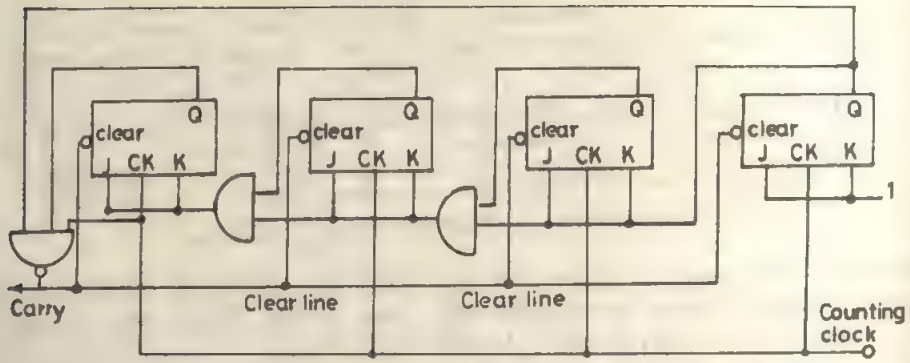



FIG. 4·19 SYNCHRONOUS DECADE UP COUNTER

### 4.4.3. Decade Counters

Decade counters are logic circuits counting numbers from 0 to 9 or from 9 to 0 Such counters are also called module 10 counters or BCD counters (binary coded decimal). A decade up counter can be constructed using a 4-bit binary counter with some modifications. In the binary counter after a count of nine, the next count is ten, while in decade counter it is zero. Thus if it is possible to enter a '0' in the binary counter at the end of the count '9', we can easily construct a decade counter. Normally most of the available F/Fs in TTL logic family have set and rest asynchronous inputs (called preset and clear respectively) and thus a state of 'nine' (1001) could be decoded and used to clear all the F/Fs of the counter as shown in Fig. 4.19.

On similar lines a decade down counter can be constructed by decoding a state of 0000 and entering 1001 in the counter through asynchronous inputs of the F/Fs.

In general a binary counter using $n$ bits could be modified to count modulo $P(P \leq 2^n)$. An $n$-bit binary counter is in fact a modulo $2^n$ counter. While counting up or down, when the state '0000' is reached it is to be indicated by the carry or borrow signal for up or down counters respectively.

### 4.4.4. Digital Clock : An Application of Counters

A digital clock using digital logic circuits

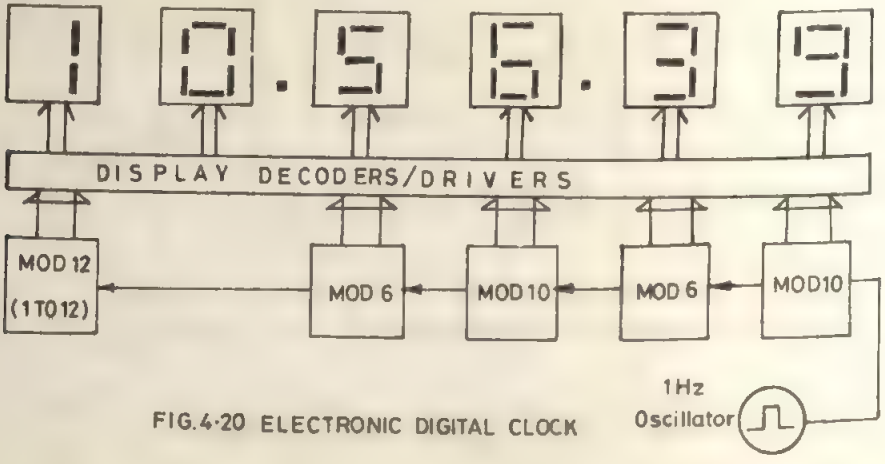



FIG. 4·20 ELECTRONIC DIGITAL CLOCK

and seven segment displays, can be constructed as discussed in this section. In the standard TTL 54/74 logic family, decade and other counters are available. We shall discuss the design using modulo $p$ counters, p taking values of 6, 9, 2 etc., depending upon the position of the counter in the counting chain. An electronic clock with a display of hours, minutes and seconds requires 2 digit counters for minutes and seconds and a special 2 digit counter counting from 1 to 12 for hours.

The complete block diagram of the clock is shown in Fig. 4 20. This clock may use a frequency of 50 Hertz (line frequency) as the input. This frequency is scaled down by a modulo 50 counter which can be constructed using 6-bit binary counter with reset at the count of 49. Every counter block gives out the carry signal which acts as a clock for the next counter block. First two modulo counters count modulo 60. The same is the case with next two counters (minutes). The last counter is a modulo 12 counter. All the mod $p$ counters used in the logic diagram assume states from 0 to $p$-1, except hour counter which counts from 1 to 12.

The contents of these counter registers are displayed using seven segment displays. If the displays do not have decoders, they will have to be inserted between the counter stages and the displays.

### 4.4.4. Binary Rate Multiplier

Modulo $p$ counters discussed in the earlier sections can count or divide the input waveforms. Suppose it is required to pass '$k$' number of pulses for every $p$ input pulses, $p>k$, to the output, with the best possible (uniform) spacing of the pulses in time. Consider the sequence of 16 input pulses as shown in Fig. 4.21 ($a$) and the output shown in Fig. 4.21 ($b$) which has 5 pulses. One could output 5 pulses in a variety of way. In Fig. 4.21 ($b$), the pulses are distributed in the best possible manner in time, while in Fig 4.21 ($c$), there is no pulse for a consecutive 11 pulse positions. Ideal equal spacing in time is impossible since we cannot produce frequencies of any arbitrary scale factor. In the above example a scaling of 5/16 is achieved on the input pulses. Such tasks cannot be done by counters alone.

Binary rate multipliers are the circuits designed to carry out this task. The name binary rate multipliers is given to these circuits since they multiply the rate by a factor $\frac{K}{2^n}$.

The logic circuit of the 4-bit binary rate multiplier is shown in Fig. 4.22 and works as foHows. The BRM of Fig. 4.22 can multiply any input frequency by a factor of $\frac{K}{2^4}$, $0< K \leqslant 15$. The required rate multiplier coefficient (K) is loaded in the 4-bit register used for the purpose. A 4-bit up counter counts the input frequency $f$. The counter bit wave forms represent the $f/2, f/4, f/8, f/16$ components.

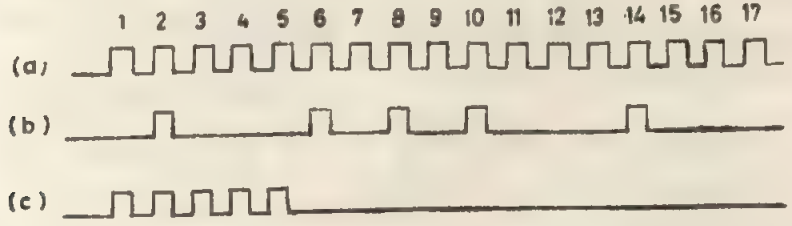



FIG. 4.21 PULSES TO ILLUSTRATE BRM

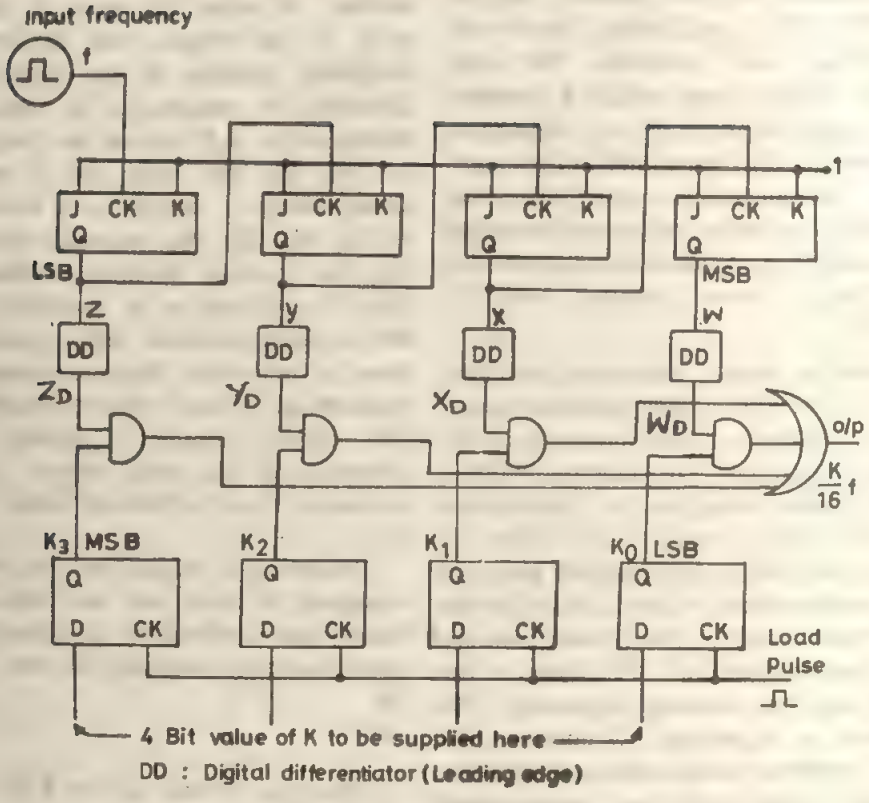



FIG. 4·22 4-BIT BRM

Considering only 0—1 type of transition 0—1 transitions occur only for one F/F at a time, the F/F Z makes exactly 8 transitions, $y$ makes 4 transitions, $x$ makes 2 transitions, and $w$ makes 1 transition for every 16 input pulses. If one wants to produce K pulses for every 16 pulses (as indicated by the coefficient register) we could select the combinations of these transitions depending upon bits of K. Thus a bit $K_3=1$ implies a number 8 and thus transitions of $z$ could be used in con junction with the F/F $K_3$. Similarly for othe bits the respective counter bits are chosen.

The number of pulses at the output can be written as Boolean function T as given below. A digital differentiator can be used to produce a pulse for a transition.

$$T=K_3 z_D+K_2 y_D\ K_1 X_D+K_0\ w_D$$

where $z_D$, $y_D$, $x_D$ and $w_D$ are the pulses produced for the transitions of $z$, $y$, $x$ and $w$ signals by differentiators.

## 4.5. TIMING AND CLOCK CIRCUITS

Timing and clock circuits are very important in digital logic circuits. Practically every logic circuit has a clock circuit. The clock pulses we talked about in the earlier sections are usually derived from the clock circuits. We discuss in this section two basic circuits called 'astable' (digital oscillator) and a monostable logic elements.

### 4.5.1. Astable (Digital Oscillator)

This circuit is designed to produce square wave oscillations. The obtained square wave is usually used as a digital system clock. Fig. 4.23 shows one implementation of the astable circuit using two NAND gates.

This circuit keeps oscillating once it is

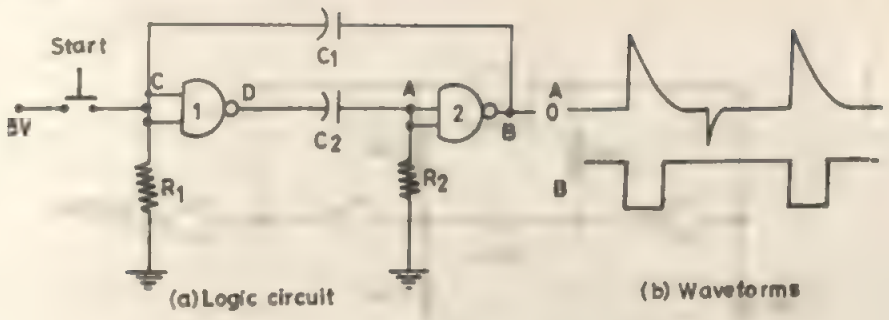



FIG.4·23 ASTABLE (DIGITAL OSCILLATOR)

started. Assuming that it oscillates, the operation can be briefly explained as follows. A 0 to 1 transition at the output of gate 1, is transmitted by the capacitor $C_2$ to the input of gate 2, which therefore produces a 1 to 0 transition at its output. This transition produces a negative voltage pulse at the inputs of the gate 1 which is obsorbed by the input diodes (in TTL) thus keeping inputs at a logical '0'. When $C_2$ gets discharged, gate 2 produces a 0 to 1 transition which has a similar effect as a 0 to 1 transition on the gate 1. Fig. 4.23 (b) shows the waveforms on the points A and B in the logic circuit of Fig. 4.23 (a).

The frequency of the oscillator depends upon the RC time constants in the circuit. The calculation can be made as follows by taking gate 2 as the output.

**The ON (Logical '0') Time**

Assuming the voltage of 0·4 volts (logic 0) at the point A, when gate 1 changes from 0 to 1 (logic 1 corresponds to 3·6 volts), the voltage at the point A decays exponentially from 4 volts towards zero. Using TTL logic gates whose transfer curve separates 1 and 0 at 1·2 and 1·4, we can say that gate 2 will change the output when its input falls to 1·2 volt. The time $t_0$ required is given by :

$$4\,e^{-t_0/R_2C_2} = 1.2$$

*i.e.*,

$$\frac{-t_0}{R_2C_2} = \log_e \frac{1.2}{4}$$

$$t_0 = R_2C_2 \log_e \frac{4}{1.2}$$

**The OFF (Logical '1') Time**

Similarly the off time (logical '0') $t_1$, depends on the other time constant and is given by :

$$t_1 = R_2C_1 \log_e\left(\frac{4}{1.2}\right)$$

The frequency of oscillation is given by $\frac{1}{t_1+t_0}$ Hz.

It is to be noted that the value of $R_2$ or $R_1$ cannot be chosen arbitrarily. This is due to the fact that the resistor must represent a sinking load, capable of sinking the specified input current of the gate without raising the voltage across itself beyond the acceptable logical '0' level (*i.e.*, 0·8 volts). The value of the resistance for TTL should not be bevond 500 ohms.

The RC oscillators built from the above principle are not stable enough in frequencies due to the variations in parameters like, currents, resistances capacitors etc., due to aging and ambient conditions.

**Crystal Oscillators**

Quartz crystals are components whose natural frequencies of oscillation could be exploited in constructing clocks. These crystals are easily available with a variety of frequencies of oscillations above 100 KHz. A crystal can be used in the clock circuit as shown in Fig. 4.24. Crystal frequency oscillators are highly stable in frequencies.

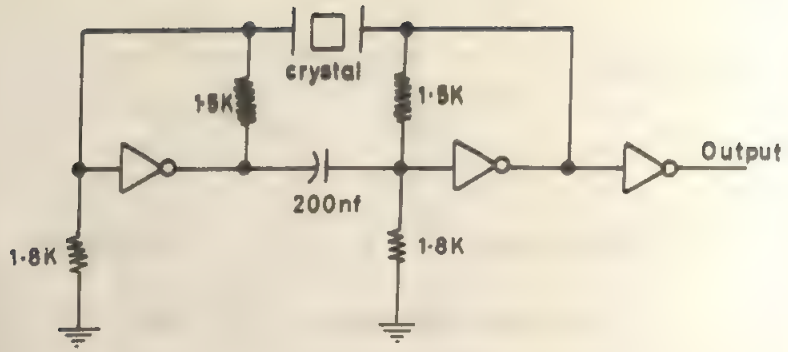



FIG. 4·24 CRYSTAL OSCILLATOR

They represent an inexpensive way of accurately timing the various events in the digital circuits.

### 4.5.2. A Monostable

A monostable is a circuit which produces a pulse of a required width when triggered for the operation. One of the simple logic circuits of a monostable is shown in Fig. 4.25. This circuit differs from the circuit of Fig. 4.23 in one of the feed back connections.

Normally the triggering signal and $\bar{Q}$ output are at logical '1' state. This makes gate 1 output normally '0'. When the triggering signal goes low, the output of gate 1 goes high and this transition is passed to the point A, making gate 2 go low. Since the output of the gate 2 is fed as input to gate 1, gate 1 output remains at high state even if the triggering signal returns to '1' state. The output of gate 2 remains low till the capacitor discharges. The time constant RC thus determines the pulse width at Q.

The logic circuits discussed in this section presented the basic principles involved in designing these circuits using logic gates. However, all these circuits are available as ICs with many other facilities included in them so that they could be used for variety of applications.

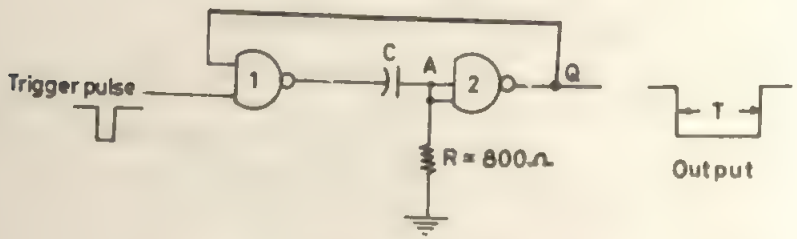



FIG. 4·25 MONOSTABLE CIRCUIT

## EXERCISES

1. You have been given the clocked synchronous edge triggered flip-flops. By addition of external logic convert them as required.

(*a*) a J-K flip-flop into D flip-flop
(*b*) a J-K flip-flop into T flip-flop
(*c*) a T flip-flop into D flip-flop
(*d*) a D flip-flop into T flip-flop
(*e*) a D flip-flop into a J-K flip-flop
(*f*) a T flip-flop into J-K flip-flop.

2. Using T flip-flops design a 4-bit shift register.

3. Using D flip-flops design a 4-bit binary counter.

4. Using D flip-flops design a decade up counter.

5. Repeat (4) for decade down counter.

6. Design a modulo 60 asynchronous counter using J-K flip-flops having preset and clear inputs.

7. Design a modulo 60 synchronous counter using J-K flip-flops.

8. Assume that the propagation delay of a flip-flop in (7) is 50 ns and delay of gates is 60 ns. Calculate the maximum clock frequency for the counter in (7).

9. Design a 5 phase clock which gives 5 pulses on lines $P_1$, $P_2$, $P_3$, $P_4$ and $P_5$. The pluses are in time sequences, *i.e* , $P_1$ comes first, then $P_2$ comes, then $P_3$, then $P_4$, then $P_5$ and then $P_1$ again. These pulses are to derived from a master clock oscillator.

10. A master clock is a free running oscillator. This clock has to be given to the circuits which are making use of it. The clock has to be stopped and started by asynchronous inputs *i.e.*, start/stop signal may come from a outside the system. Design a circuit which would synchronise the start/stop operation such that the clock will be started/stopped in synchronism.

11. Design a sequential circuit which will take 1's complement of an incoming serial binary number (sign bit is leading, followed by other bits in least to most significant order).

12. Repeat (11) for 2's complement.

13. Design a shift left register which can shift the data by 1 bit, 2 bit or 4 bits as per the requirement. Assume three control signals which will indicate the required shifting. Assume that only one of these three signals will be true at a time, *i.e.*, we require one of the available three shift operations at a time.

14. Repeat (13) for right shift.

15. Repeat (13) for left as well as right shift. Assume extra input indicating left or right.

16. Design a shift register which can shift left or right by any amount from 1 to 7. The number of shifts required will be told through a 3-bit binary number. (Hint. Use (13), (14), (15) and 3 pulses).

17. A shift register is given. Construct a circuit which tells the number of 1's contained in this shift register (assume 16-bit shift register).

18. Shift register bits in (17) are numbered from 0 through 15 and it contains exactly a single '1' in it. It is required to know the position of this '1'. Construct a circuit to carry this out.

19. A 16-bit register contains 1's and 0's. Design a logic circuit to find the block which contains the maximum number of 1's. Circuit should give out the number of 1's and the starting bit from where this block starts. Register can be assumed as a shift register if required.

20. Using 4-bit binary up/down counters, design a circuit which (*i*) adds, (*ii*) subtracts two unsigned binary numbers. Give the maximum number of pulses required.

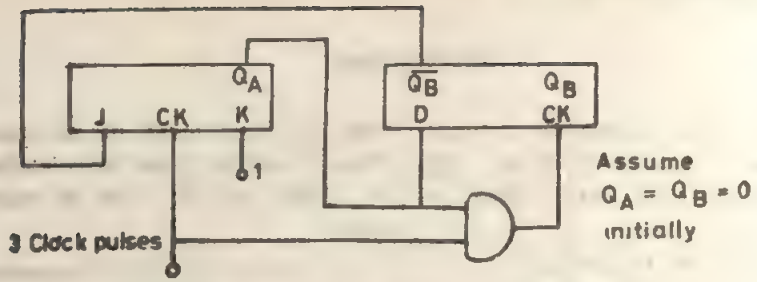



FIG. 4.26. EXAMPLE CIRCUIT

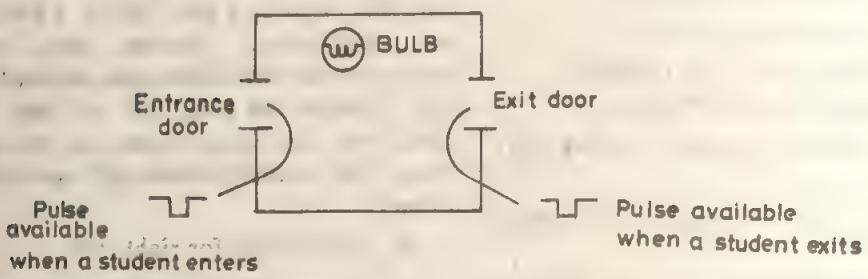



FIG. 4.27 STUDY ROOM

21. Using the adder designed in (20), design a multiplier to multiply 4-bit numbers.

22. Using counters, design a logic circuit which finds remainder of a number X when it is divided by Y. Assume X as 8-bit and Y as 4 bit long. Also assume that X and Y are unsigned numbers.

23. Repeat (20), (21) and (22) for decimal numbers using decade counters.

24. Design an electronic circuit to carry out some job in your house.

**Example.** Design a circuit which will automatically switch on your study room lamp when you say 'Light' (or some word) four times and switch it off if you say 'light' twice.

25. Find the outputs $Q_A$ and $Q_B$ after applying 3 clock pulses to the logic shown in Fig. 4.26.

26. Fig. 4.27 illustrates the study room of two students. At each door, there is a light cell which outputs a pulse when a student passes through a door. Assuming that one door is for entry and other for exit, design a logic circuit, which would put ON the bulb in the room, if at least one student is in the room, else, the bulb should be OFF.

27. The pulse sequences (Fig. 4.28) $P_1$ and $P_2$ are required for certain control functions. Design the logic to obtain these sequences. Assume that the main clock (square wave oscillator) is available.

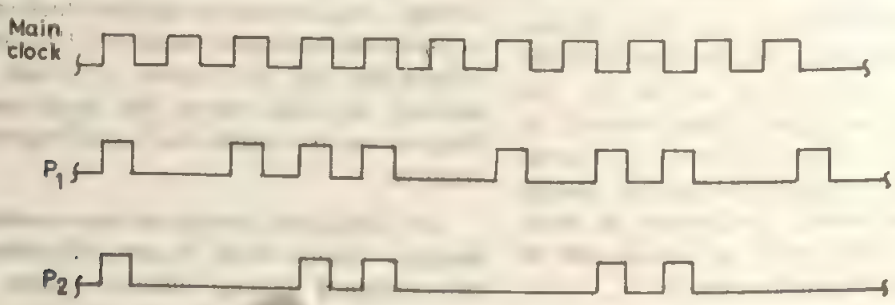



FIG. 4.28 PULSE SEQUENCES

# 5

# BINARY ADDITION AND SUBTRACTION

EVERY digital computer has in its arithmetic unit, an adder for carrying out addition/subtraction of binary numbers. In this chapter, we shall discuss how these operations are mechanised by logic circuits. Arithmetic values (0, 1) assumed by digits in binary numbers are assumed to be represented by truth values of these digit variables.

## 5.1. ADDITION OF UNSIGNED INTEGERS

Let X and Y be two n-bit binary integers to be added and let S be their sum. Thus if,

$$X = \sum_{i=0}^{n-1} x_i 2^i \text{ and}$$

$$Y = \sum_{i=0}^{n-1} y_i 2^i, \text{ then}$$

$$S = \sum_{i=0}^{n-1} s_i 2^i,$$

where S is the sum with $s_i$ as its $i$th bit.

**Example 5.1.**

Add X=01011 and Y 01011

43210 ←digit positions

X=01011

Y=01001

10110 ←carries

S=10100

In this example, at every digit position $i$, $i=1, 2 \ldots n-1$ we have added three single bit numbers $x_i$, $y_i$ and $c_{i-1}$ (carry). The result of this addition gave us a 2 bit number, whose least significant bit is put as a sum digit in the position $i$ and the most significant bit is passed to the next stage as a carry, for example, at bit position 1, we have :

$$x_1+y_1+c_0=1+0+1=10$$

carry ↑ ↑ sum

The example given above clearly identifies a primitive block (adder for $x_i$, $y_i$ and $c_{i-1}$) which is functionally complete to implement the addition of n-bit numbers. This block is called a 'full Adder' and its logic circuit implementation is discussed in the next subsection.

### 5.1.1. Full Adder

The block diagram of a full adder is shown in Fig. 5.1. To construct the logic circuit for this block, we use Boolean algebra and work out the multi-output switching function for sum and carry outputs. Truth functions for sum and carry are shown in Table 5.1.

TABLE 5.1. Full Adder

| $x$ | $y$ | $c_{in}$ | $s$ | $c$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

The output Boolean functions $s$ (sum) and $c$ (carry) are given by :

$$s = x'y'c_{in} + x'yc'_{in} + xy'c'_{in} + xyc_{in}$$
$$= c_{in}(x'y' + xy) + c'_{in}.(x'y + xy') \cdot$$
$$= c_{in} \oplus x \oplus y \text{ ; and}$$
$$c = x'yc_{in} + xy'c_{in} + xyc_{in}' + xyc_{in}$$
$$= (x \oplus y)\, c_{in} + xy$$

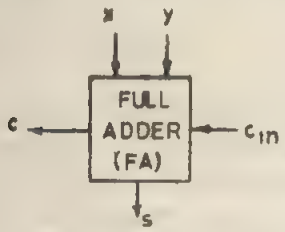



FIG. 5·1 BLOCK DIAGRAM

Also,

$$c = (x+y)\, c_{in} + xy.$$

Logic network implementing these functions is shown in Fig. 5.2.

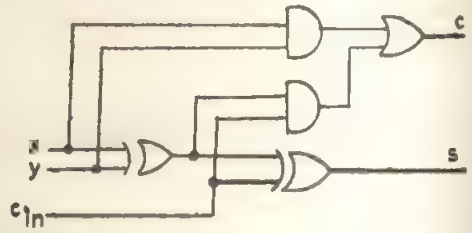



FIG 5·2 LOGIC CIRCUIT

### 5.1.2. Parallel Addition of Two Numbers

The addition of two n-digit numbers can be carried out now, using one full adder for each stage by arranging $n$ full adders as shown in Fig. 5.3.

Note that carries propagate (ripple) from least significant position towards the most significant position. Due to this reason these adders are called ripple carry adders. Assuming a time delay of $\Delta t$ units for each full adder to produce a carry and a sum, this logic network can perform the addition in $\Delta t.n$ units of time.

As an integrated circuit package, 4-bit adders are commonly available. One such module is 7483 in the 7400 series of TTL digital ICs. This logic module contains four full adders connected to form a 4-bit binary adder, with a special logic to work out the carry from the most significant bit position.

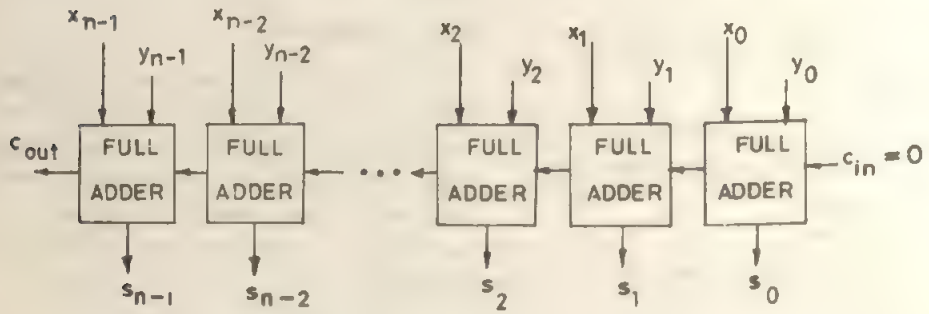



FIG.5·3 n-BIT PARALLEL RIPPLE CARRY ADDER

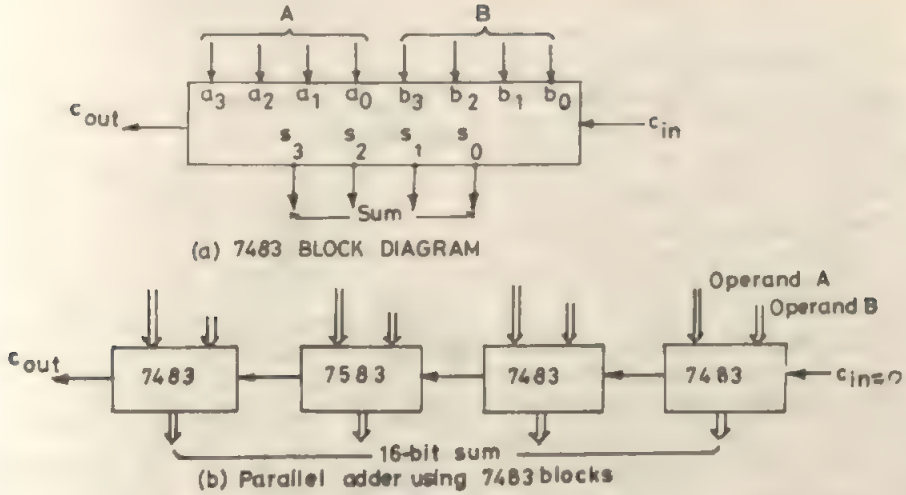



FIG.5·4 ADDER CIRCUIT

The block diagram of 7483 is shown in Fig. 5 4 (*a*), and a 16-bit binary adder using these modules is shown in Fig. 5.4 (*b*).

Due to the special carry out (see section 5·6) circuit within 7483, the time delay of $c_{out}$ signal is only one unit (instead of four units as would normally be expected). Therefore the addition time for the 16-bit adder of Fig. 5.4 (*b*) is only four units of time.

### 5.1.3. Serial Addition of Two Numbers

The bit by bit serial addition of numbers can be carried out using only one full adder. The arrangement is shown in Fig. 5 5. In this diagram, two n-bit shift registers hold the two operands to be added. Initially carry F/F is cleared by the reset signal. The clock is started to give n-clock pulses. At every stage, the full adder works out the sum and

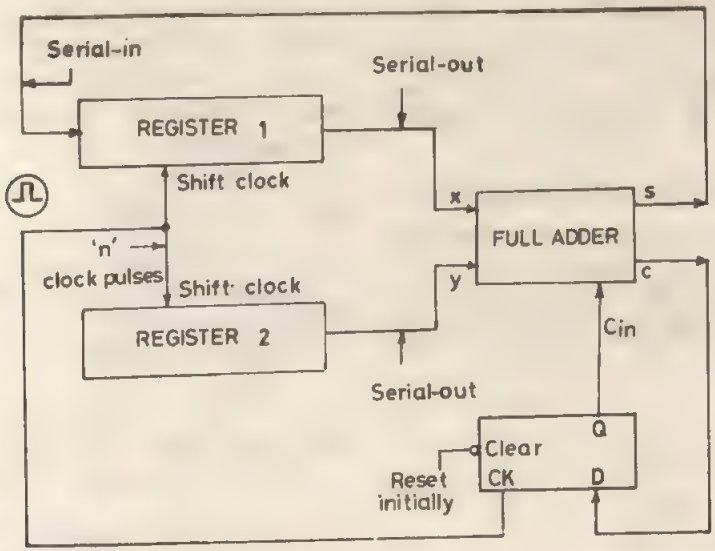



FIG. 5·5 n- BIT SERIAL BINARY ADDER

carry signals. The sum is connected to serial-in point of one of the register and carry is stored in carry F/F ; which supplies the carry for next stage. After n-clock pulses, the register 1 contains the sum and carry F/F contains the carry out from the last stage. Register 2 could be restored with original number by providing the serial-out to serial-in connection (not shown in Fig 5.5).

## 5.2. BINARY SUBTRACTION

A single stage subtractor ($s_i = x_i - y_i - b_{i-1}$) could be developed on the lines of full adder. To carry out subtraction of n-bit numbers, $n$ such blocks could be used, however due to properties of complements, subtraction could be very easily expressed as an addition of the complement and therefore, arithmetic units seldom have a subtraction hardware. This point will be apparent from section 5.3.

## 5.3. ADDITION AND SUBTRACTION OF SIGNED BINARY NUMBERS

Signed binary numbers could be represented in any of the following forms (see Chapter 2),

1. Sign-magnitude form.
2. Sign-2's complement form.
3. Sign-1's complement form.

In this section mechanized methods for addition and subtraction of numbers are presented for each of the above number representations.

### 5.3.1. Numbers in Sign-Magnitude Form

Let X and Y be two signed numbers with signs $x_s$ and $y_s$, and magnitudes X* and Y* respectively. The addition/subtraction of X and Y have four possible cases as shown below :

(*a*) **Addition**

i. (+X*)+(+Y*)
ii. (+X*)+(−Y*)
iii. (−X*)+(+Y*)
iv. (−X*)+(−Y*)

(*b*) **Subtraction**

i. (+X*)−(+Y*)
ii. (+X*)−(−Y*)
iii. (−X*)−(+Y*)
iv. (−X*)−(−Y*)

These four cases of subtraction are equivalent to :

(+X*)+(−Y*) (Case ii)
(+X*)+(+Y*) (Case i)
(−X*)+(−Y*) (Case iv)
(−X*)+(+Y*) (Case iii).

These four cases are the same as those listed for the addition. Thus, a subtraction (X−Y) of signed numbers is equivalent to addition of Y with complemented sign.

Let $s_s$ and S* be the sign and magnitude of the sum S=X+Y. Clearly S* could be obtained by either addition or subtraction of unsigned numbers X* and Y*. The sum S* and sign $s_s$ for cases (*i*) and (*iv*) is given by :

$S^* = X^* + Y^*$ and
$s_s = x_s$ (or also $y_s$)

For the remaining two cases, the sum is given by :

$$S^* = \begin{cases} X^* - Y^* & \text{(Case ii)} \\ Y^* - X^* & \text{(Case iii), and} \end{cases}$$

the sign of the result (sum) will be positive if the difference is positive. On the other hand if the difference is negative, the sign of sum will not only be negative, but the result will be in 2's complement form. This point is illustrated by the following example :

**Example 5.2.**

X=+5, Y=−7
thus X=0, 0101
Y=1, 0111

Addition of X and Y is performed as below :

| | | |
|---|---|---|
| $X^*=$ | 0101 | +(+5) |
| $-Y^*=$ | 0111 | +(−7) |
| | 1 1100 | borrows |

$S^* = (-1)\ 1110 - 1.2^4 + 14 = -2$

borrow

Because of the borrow at the most significant position (its value is $-1.2^4$), the result

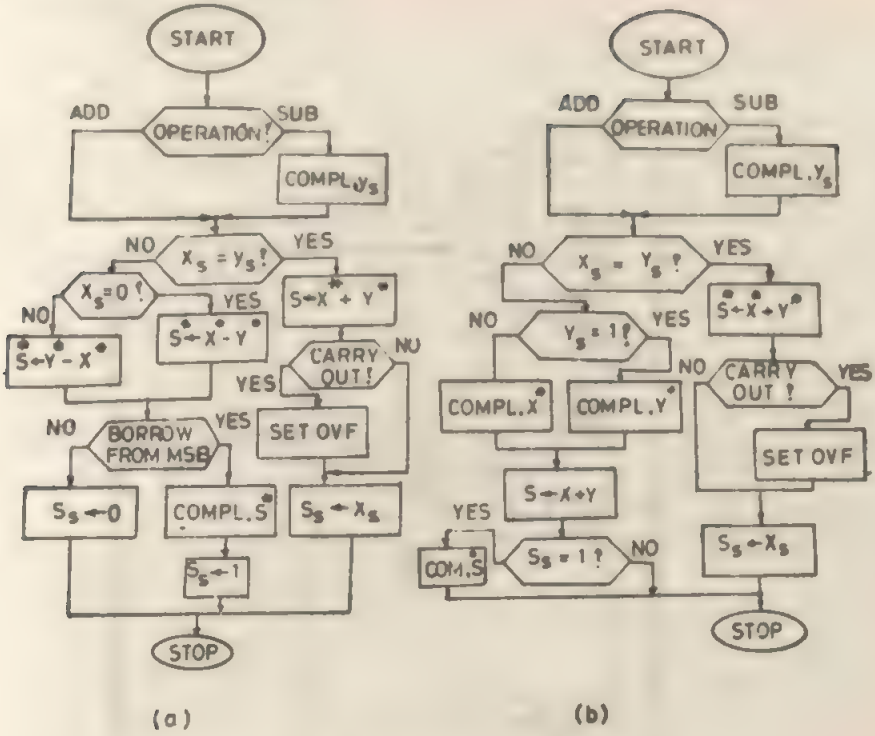



FIG.5·6 ADDITION/SUBTRACTION OF NUMBERS IN SIGN MAGNITUDE FORM

is $-1.2^4+14=-2$ (0010 in binary); the S* obtained (1110) is in 2's complement form. Therefore, S* must be complemented so that sum is obtained in sign-magnitude form.

For certain values of operands, the result of an arithmetic operation is not possible to be stored in a machine register (usually fixed in length). These cases are indicated by setting a flag (a F/F) called overflow (OVF). Note that a carry/borrow from the most significant position does not always result in a overflow.

A complete algorithm for addition and subtraction of numbers in sign-magnitude form is given in Fig. 5.6 (*a*).

The implementation of the actions of the flow-chart of Fig. 5.6 (*a*) by digital logic requires an adder and two subtractors (one subtractor if X*, Y* are appropriately exchanged). In addition, we also require a decoding logic to decide the various combinations of $x_s$ and $y_s$, so that the appropriate flow of actions are performed. A reduction in hardware is possible if complements are used. For example, let $X=+X^*$ and $Y=-Y^*$; then X+Y is given by :

$$(+X^*)+(-y^*)=+X^*+(\overline{Y^*}).$$

Now both operands could be treated as unsigned, and an addition performed on X and Y. [See Fig. 5 6 (b)]

**Example 5.3.**

Add X=+0101 and Y=−0111 given in signmagnitude form.

X=0,0101
Y=1,0111 ; $\overline{Y^*}$=1001

Hence,

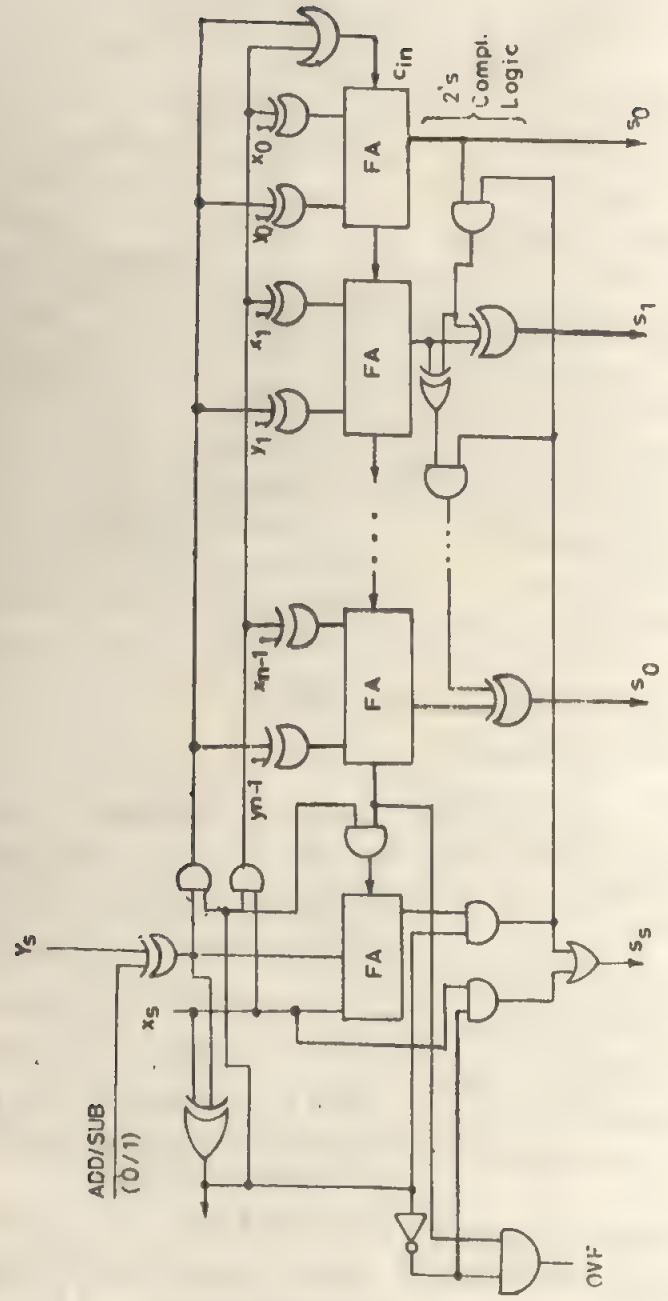



FIG. 5·7 ADDER/SUBTRACTOR IMPLEMENTATION OF FLOWCHART OF FIG. 5·6 (b)

$$\begin{array}{rl} & X = 0,0101 \\ y_s, & \overline{Y^*} = 1,1001 \\ \hline & 1,1110 \end{array}$$

sum

The result is in the 2's complement form, because sign bit of the result is negative. Only for cases (ii) and (iii), a complement of negative operand need be taken. With these modifications a new flow chart is shown in Fig. 5.6 (*b*), and its logic implementation is shown in Fig. 5.7. Full adder outputs are fed to the parallel 2's complementing circuit.

### 5.3.2. Numbers in Sign -2's Complement Form

Recall that (Chapter 2) sign-2's complement representation of binary numbers uniformly represents the positive as well as negative numbers, as a polynomial of an unsigned number [equation (2.1)] with sign digit having negative weight. Therefore, four cases of section 5.3.1 of addition/subtraction can be uniformly treated with only addition, where complete signed numbers take part in the addition here. Even for sign bit (which has negative weight), the usual rules of binary addition could be applied. This is so because the sum or difference table, for three single bit numbers is same. The addition flow charts of Fig. 5.6 reduces to the one shown in Fig. 5.8. Here an OVF results if two numbers have same sign and the sign of result is different from that of the operands. Fig. 5.9 shows the 16-bit adder implementation of the flow-chart of Fig. 5.8 using 7483 ICs.

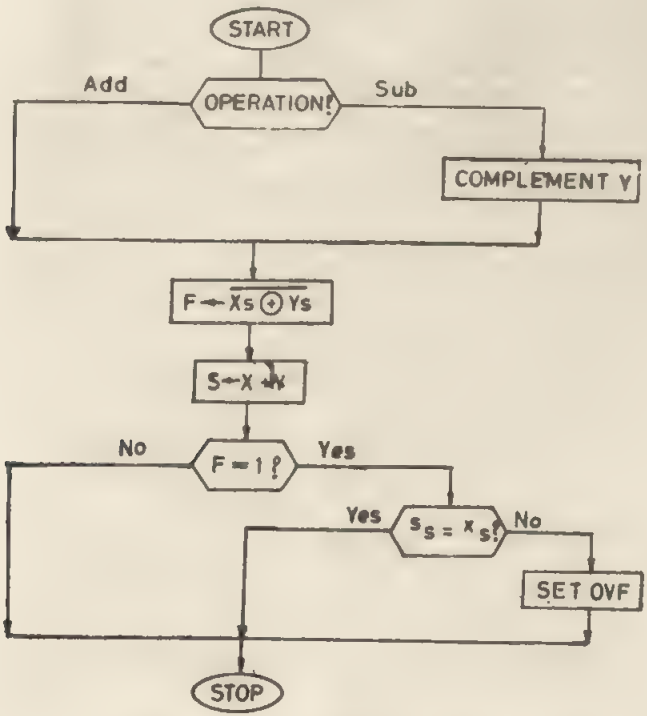



FIG. 5·8 ADDITION / SUBTRACTION OF NUMBER IN SIGN-COMPLEMENT FORM.

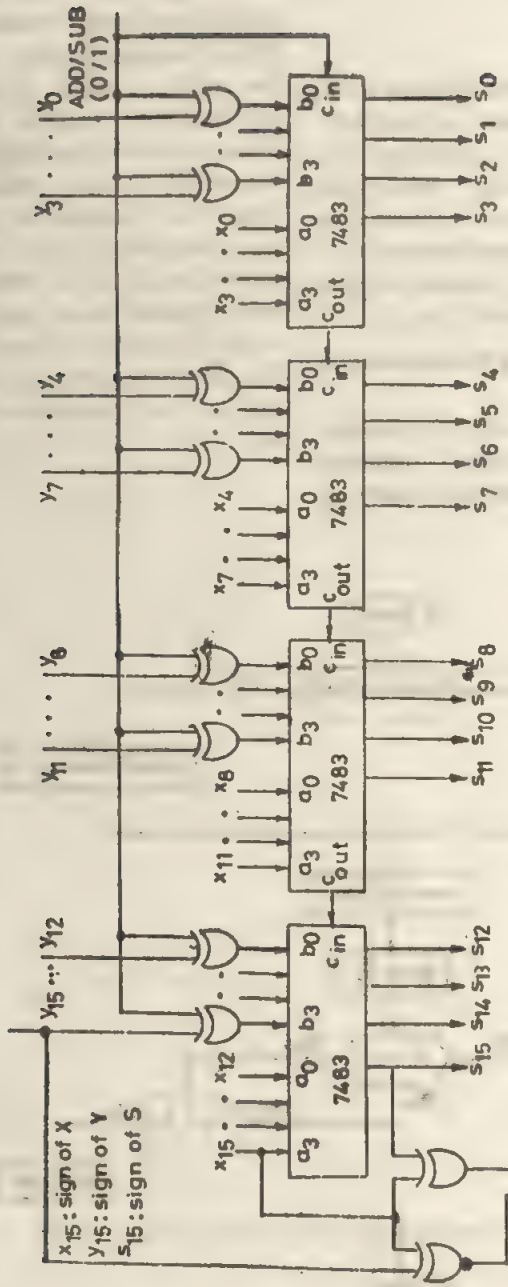



FIG. 5·9 16-BIT ADDER/SUBTRACTOR FOR NUMBERS IN COMPLEMENT FORM

**Example 5.4.** X and Y in each of the following are in sign-magnitude form. Convert then in sign-2's complement form and perform the addition.

(1) X=+0101, Y=+0111

X and Y are first represented in sign-complement form, and simply added as if they were unsigned numbers.

*i.e.*, X= 00101 +5
+Y= 00111 +7
S= 01100 +12
↑
sign bit

$OVF=(x_s \oplus y_s).(x_s \oplus s_s)$
$=\overline{(0 \oplus 0)}\ (0 \oplus 0)$
$=1 \cdot 0$
=0 (OVF=0 means no OVF and result is correct)

(2) X=+0101, Y=−0111

X= 00101 +5
+Y= 11001 −7
S= 11110 −2
↑
sign bit

$OVF=(x_s \oplus y_s)\ (x_s \oplus s_s)$
=0 (hence the result is correct)

(3) X=−0101, Y=+0111

X=11011 −5
+Y=00111 +7
1 carry in MSB
S=1 00010 =+2
↑ ↑
ignore— sign bit

$OVF=\overline{(x_s \oplus y_s)}.(x_s \oplus s_s)=0$

carry from sign bit is ignored, because actually there is no carry *i.e.*, $x_s+y_s+c_{in}$ $=-1+0+1=0$.

(4) X=−0101, Y=−0111

X=11011
+Y=11001
1 10110←carries
S=1 10100
↑ ↑
Ignore— sign bit

carry from sign bit ignored (actually there is no carry *i.e.*, $x_s+y_s+c_{in}=-1+-1+1=-1$)

$OVF=\overline{(x_s \oplus y_s)}.(s_s \oplus x_s)$
$=\overline{(1 \oplus 1)}.(1 \oplus 1)$
$=0.$

In none of these cases an overflow resulted, because sum could be accommodated in four bits. To illustrate the OVF logic, we shall work out cases 2 and 4 using four bit numbers.

(5) X=+101 Y=−111
X=0101 +5
+Y=1001 −7
S=1110 −2
↑
sign

$OVF=\overline{(x_s \oplus y_s)}.(x_s \oplus s_s)$
$=\overline{(0 \oplus 1)}\ (0 \oplus 1)$
=0. (No OVF, hence result is correct)

(6) X=−101, y=−111
X=1011 −5
+Y=1001 −7
S=0100 +4
↑
sign

$OVF=\overline{x_s \oplus y_s}.(x_s \oplus s_s)$
$=\overline{(1 \oplus 1)}.(1 \oplus 0)$
=1, (OVF, hence result is wrong).

The overflow results because 12 cannot be stored in a 3-bit register. This fact is recognised by OVF logic and an indication is given by setting an OVF flag.

### 5.4.3. Numbers in Sign -1's Complement Form

A 1's complement of an integer is smaller than its 2's complement by 1 in LSB. Hence negative numbers in sign-1's complement form will be smaller than those in 2's complement form by 1. Noting this fact, rules for 1's complement addition/subtraction could be formulated. It can be proved that signed numbers in 1's complement form could be added as if they were unsigned numbers ; but a carry out from sign bit is to be added back to the least significant position.

**Example 5.5.** (1) X=+5, Y=−7,
(2) X=−5, Y=−7

In sign 1's complement form these are written and added as follows ;

| | | | |
|---|---|---|---|
| (1) | X | 0,0101 | +5 |
| | +Y | 1,1000 | −7 |
| | | 1,1101 | −2 |
| (2) | X | 1,1010 | −5 |
| | +Y | 1,1000 | −7 |
| End around carry | ← | 1,0010 = | −13 |
| | + | 1 | +1 |
| Result | | 1,0011 = | −12 |

## 5.5. CARRY LOOKAHEAD ADDERS

The parallel $n$-bit adder (ripple carry adder) discussed earlier requires carries to propagate through the various stages. Worst case of carry propagation is a carry $c_{in}$ propagating to the last stage. This involves a delay of $n$ time units, and for large $n$, this time delay may be prohibitive. In this section we shall discuss a technique called carry lookahead addition which makes practicable the implementation of adders requiring $\log(n)$ time units for $n$-bit addition.

### 5.5.1. Basic Principle of Carry Lookahead Addition

In this technique all the carries are anticipated in advance so that a particular stage could generate a sum independently of the carries from previous stages. This is illustrated below :

Consider a ripple carry adder shown in Fig. 5.3.

Basic carry equation of the full adder is given by :

$$c=(a+b)c_{in}+a.b$$

Thus $c_0=(a_0+b_0)c_{in}+a_0b_0$

$c_1=(a_1+b_1)c_0+a_1b_1$

Let $p_i=a_i+b_i$ and

$g_i=a_i\,b_i$ then,

$c_0=p_0c_{in}+g_0$

$c_1=p_1c_0+g_1$, substituting for $c_0$ from above equation, we get

$$c_1=p_1(p_0c_{in}+g_0)+g_1$$
$$=p_1p_0c_{in}+p_1g_0+g_1,$$ proceeding similarly, we get

$$c_2=p_2p_1p_0c_{in}+p_2g_1+p_2p_1g_0+g_2$$
$$c_3=p_3p_2p_1p_0c_{in}+p_3p_2p_1g_0+p_3p_2g_1+p_3g_2+g_3.$$

In general a carry from $n^{th}$ stage is given by :

$$\begin{aligned}c_n=&p_n\,p_{n-1}\,p_{n-2}\,..p_2p_1p_0c_{in}\\&+p_n.p_{n-1}...p_2p_1g_0\\&+p_np_{n-1}...p_3p_2g_1\\&+p_np_{n-1}...p_4p_3g_2\\&+\\&.\\&.\\&.\\&+p_n\,p_{n-1}g_{n-2}\\&+p_ng_{n-1}\\&+g_n\end{aligned}\qquad ...(5.1)$$

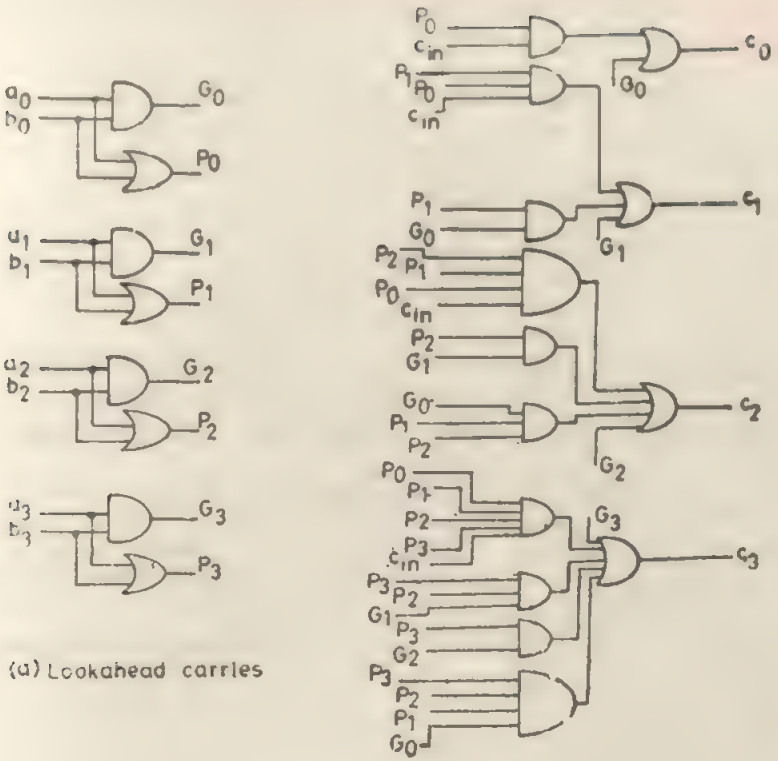
Let $P=p_n.p_{n-1}...p_3p_2p_1p_0$

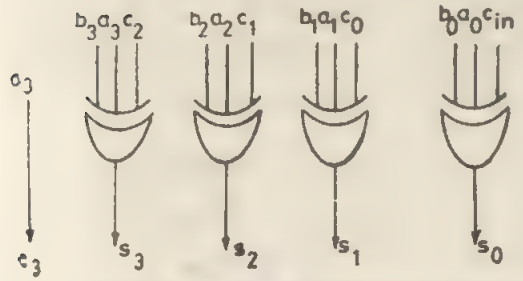
and $G=p_np_{n-1}...p_3p_2p_1g_0$
$+p_n.p_{n-1}...p_3p_2g_1$
$+p_np_{n-1}...p_4p_3g_2$
$+$
$.$
$.$
$.$
$+$
$+p_n\,g_{n-1}$
$+g_n$

Then $c_n=P.c_{in}+G$ ...(5.2)

Equation (5.2) tells us whether a carry at $n^{th}$ stage will occur or not. As can be seen $c_n=1$ if G is 1, or $p=1$ and there exists an input carry $c_{in}$. This equation is not different from a full adder carry equation. G and P are the group carry generate and propagate function for a $n$-bit group.

**Note that G and P are functions of only operand bits.** Thus final carry $c_n$ depends only on operands and $c_{in}$ (carry for the very first stage), thus giving it a complete independence from previous other carries.




(b) 4-bit adder with lookahead carries

FIG.5.10 4-BIT CARRY LOOKAHEAD ADDER

Figs. 5.10 (*a*) and (*b*) show a logic circuit of a 4-bit carry lookahead adder. The carry equations are :

$$c_0 = P_0 c_{in} + G_0$$
$$c_1 = P_1 P_0 c_{in} + P_1 G_0 + G_1$$
$$c_2 = P_2 P_1 P_0 c_{in} + P_2 G_1 + P_2 P_1 G_0 + G_2$$
$$c_3 = P_3 P_2 P_1 P_0 c_{in} + P_3 P_2 P_1 G_0 + P_3 P_2 G_1 + P_3 G_2 + G_3.$$

All of these equations can be implemented using 2 level logic networks (every output signal can be realised with at the most two gate delays). Hardware complexity of two level logic for carries increases exponentially with $n$, prohibiting the use of this technique for large $n$ (greater than say 8). But there exists a way out of this if we sacrifice some

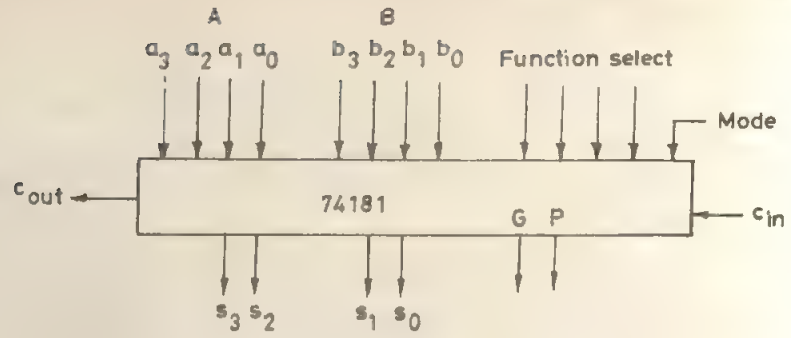



FIG. 5-11 74181 FUNCTIONAL PIN DIAGRAM

speed. This is possible by having a multilevel lookahead. There are two TTL ICs, 74181 (4-bit ALU) and 74182 (4-bit carry lookahead generator), which are designed so as to be used in the multilevel lookahead manner. This principle can be best illustrated by studying these ICs and their usage.

TABLE 5 2 ; 74181 Functional Description

| *Function Select* | | | | | *Output Function (No incoming carry)* |
|---|---|---|---|---|---|
| *S3* | *S2* | *S1* | *S0* | *Logic Functions* | *Arithmetic Operations* |
| 0 | 0 | 0 | 0 | $F=\bar{A}$ | F=A |
| 0 | 0 | 0 | 1 | $F=\overline{A+B}$ | F=A+B |
| 0 | 0 | 1 | 0 | $F=\bar{A}B$ | $F=A+\bar{B}$ |
| 0 | 0 | 1 | 1 | F=Logical 0 | F=minus 1 (2's complement) |
| 0 | 1 | 0 | 0 | $F=\overline{AB}$ | F=A plus AB |
| 0 | 1 | 0 | 1 | F=B | F=[A+B] plus AB |
| 0 | 1 | 1 | 0 | $F=A\oplus B$ | F=A minus B minus 1 |
| 0 | 1 | 1 | 1 | $F=A\bar{B}$ | F=AB minus 1 |
| 1 | 0 | 0 | 0 | $F=\bar{A}+B$ | F=A plus AB |
| 1 | 0 | 0 | 1 | $F=\overline{A\oplus B}$ | F=A plus B |
| 1 | 0 | 1 | 0 | F=B | $F=[A+\bar{B}]$ plus AB |
| 1 | 0 | 1 | 1 | F=AB | F=AB minus 1 |
| 1 | 1 | 0 | 0 | F=Logical 1 | F=A plus 2A |
| 1 | 1 | 0 | 1 | $F=A+\bar{B}$ | F=[A+B] plus A |
| 1 | 1 | 1 | 0 | F=A+B | $F=[A+\bar{B}]$ plus A |
| 1 | 1 | 1 | 1 | F=A | F=A minus 1 |

### 5.5.2. Four Bit Arithmetic Logic unit (74181)

**Brief Description** : This integrated logic circuit block is designed to carry out 32 functions of two 4-bit operands A and B. This block can perform 16 arithmetic and 16 logical functions. The four function control inputs decide one of these 16 functions, while mode select input decides the arithmetic or logical mode. Table 5.2 gives the complete functional description of this module, while Fig. 5.11 shows its block diagram.

All the carries are implemented using carry lookahead equations of Section 5.5.1. G and P are the group carry generate and propagate signals for the four bit group. All the signals are computed approximately in 12 ns.

### 5.5.3. Four Bit Carry Lookahead Generator (74182)

This logic circuit accepts group G and P signals of four groups in order of their significance. Three lookahead carries are computed using these G and P signals and carry-in signal. The outputs G and P provide the group generate and propagate signals representing a larger group composed of the four input groups. Fig. 5.12 shows the block diagram of this logic module.

The logic equations for $c_z$, $c_y$, $c_x$, G and P outputs are :

$$c_z = P_0 c_{in} + G_0$$
$$c_y = P_1 P_0 c_{in} + P_1 G_0 + G_1$$
$$c_x = P_2 P_1 P_0 c_{in} + P_2 P_1 G_0 + P_2 G_1 + G_2$$
$$P = P_3 P_2 P_1 P_0$$
$$G = P_3 P_2 P_1 G_0 + P_3 P_2 G_1 + P_3 G_2 + G_3.$$

The logic modules 74181 and 74182 can be used to design arithmetic logic units with high speed arithmetic capabilities.

### 5.5.4. Design of Adders With Multilevel Lookahead Carries

Four 74181's could be cascaded as shown in Fig. 5.13. This adder has group carries connected in a ripple carry fashion, while carries within the group are lookahead carries. The time required for addition, thus will be four times that of 74181 IC. Assuming a unit delay for each 74181 block, a 16-bit adder of Fig. 5.13 requires 4 units, while the adder of Fig. 5.3 requires 16 units. The adder of Fig. 5.13 does not use multilevel lookahead feature, but uses group lookahead and ripple carry (between groups) principle. Another way of designing a 16-bit adder is shown in Fig. 5.14. This design uses carries with two levels of lookahead. In both the circuits of Figs. 5.13 and 5.14, operands,

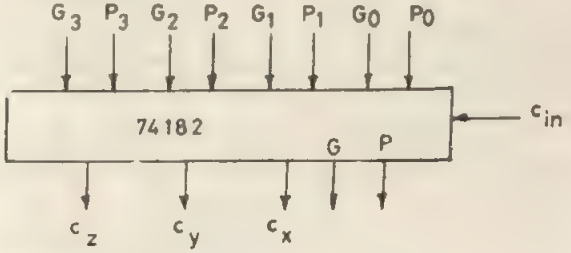



FIG. 5-12 74182 FUNCTIONAL PIN DIAGRAM


FIG. 5-13 16-BIT ADDER

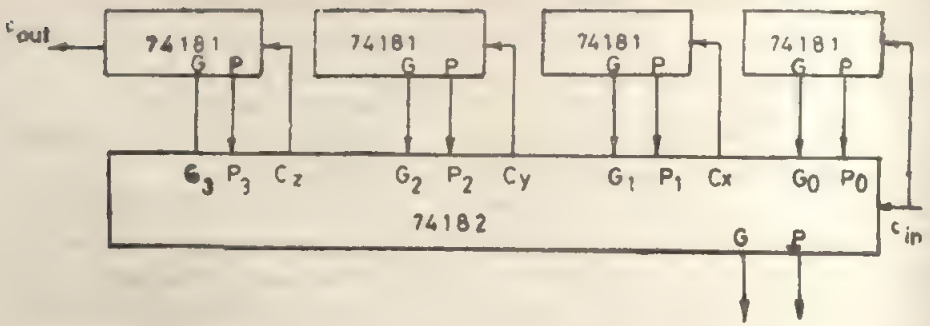



FIG 5-14 16-BIT ADDER USING TWO LEVEL LOOKAHEAD

function select and function output lines of 74181 chips are omitted. The adder of Fig. 5.14 requires 3 units of time for addition. G and P signals of 74182 represent a group generate and propagate signals for a 16-bit group. This fact can be used to build adders with additional delay of 2 units, but four times larger in size.

Fig. 5.15 shows a 64-bit adder which utilises three levels of lookahead. This adder makes use of four blocks of Fig. 5.14 (say Block A) and one additional 74182 IC.

Blocks marked A are 16-bit adders of Fig. 5.14. The time for addition of this adder will be 2 additional units over the time of the adder shown in Fig. 5.14, thus requiring 5 units. Similarly 256 bit adder could be designed using four levels of lookahead. This will require 7 units of time for addition.

### 5.5.5. Time Required by Multilevel Lookahead Carry Adders

In this section, we shall formulate the expression for time units required for addition as a function of $n$.

For generality let us have $k$-bit groups, *i.e.*, we have a $k$-bit lookahead adder and a $k$-bit carry lookahead generator designed on the lines of 74181 and 74182 respectively. Let the time delay of each of these $k$ bit blocks be $t_a$.

Let L be the number of lookahead levels required to implement the adder of $n$ bits.

Then,

$L = [\text{Log}_k n]$, where $[x]$ = smallest integer $\geqslant x$.

To evaluate the time required for addition, consider how the G and P signals propagate through various lookahead levels. At each

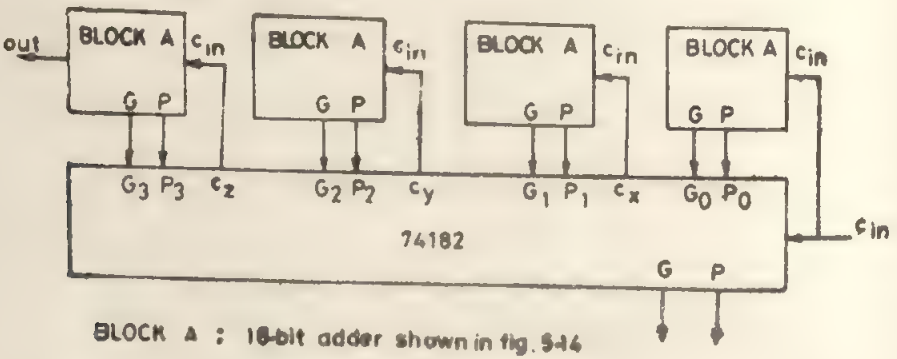
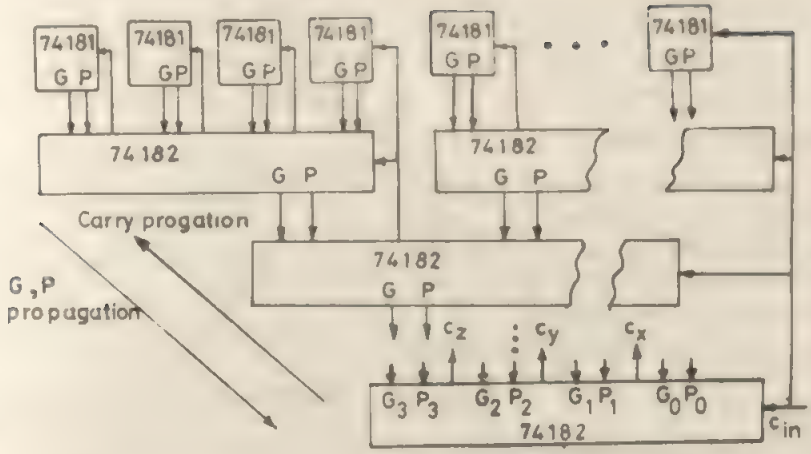



FIG.5-15 64-BIT ADDER USING THREE-LEVEL LOOKAHEAD




3.516 G,P AND CARRY PROPAGATION SCHEMATIC.

level G and P signals are calculated using the G, P signals of the previous level (for first level operand bits are used).

The settling time for $G_L$ and $P_L$ (G and P signals of Level L) will be thus L time units. At this time carries of last level are also computed. Carries in turn propagate towards level 1 and the first level produces sum, thus requiring L-1 time units. (See Fig. 5.16).

The addition time thus equals time of G, P propagation towards last level, and time taken by carries to propagate to the 1st level.

Thus addition time $T_{add} = L+L-1 = 2L-1 = (2[\text{Log}_k n]-1)$. $t_d$ units.

## 5.6 ADDERS USED IN SECOND GENERATION MACHINES

The addition techniques and adders discussed in earlier sections were seldom used in second generation computers (computers built using transistors). These computers used to have a logic wired between the accumulator (say A Register) and operand register (say B register). We shall discuss in this section how the addition used to be performed in these computers.

Let A and B be two registers holding the operands and let the result of the operation be placed back in A register. The $i$th sum bit $s_i$ is given by :

$$s_i = a_i \oplus b_i \oplus c_{i-1}$$

This equation instead of implementing using full adder logic can be alternatively implemented in two sequential steps, and therefore the method is known as **2-Step addition** method.

Step 1 : $r_i \leftarrow a_i \oplus b_i$

Step 2 : $s_i \leftarrow r_i \oplus c_{i-1}$

Now, if the result of each of these steps is put back in the A register itself, we have :

Step 1 : $a_i \leftarrow a_i \oplus b_i$

Step 2 : $a_i \leftarrow a_i \oplus c_{i-1}$

**Step 1 Implementation Logic**

Implementation of step 1 is done by a clock pulse $P_1$ and the transition table for each bit $a_i$ of A is shown in Table 5.3.

TABLE 5.3 : Next State of F/F, of A Register (after pulse $P_1$)

| $a_i$ | $b_i$ | $a_i$ (new) |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

By observing the Table 5.3, it is easy to recognise the fact that step 1 could be implemented by complementing the bits of A if the corresponding bits of B are 1.

**Step 2 Implementation Logic**

Before the step 2 begins (pulse $P_2$ arrives) every bit $a_i$ of A represents the value $a_i \oplus b_i$ on old values of $a_i$ and $b_i$ (note that the old value of $a_i$ is lost). We have partially implemented the full adder equation $s_i = a_i \oplus b_i \oplus c_{i-1}$ *i.e.*, $a_i \oplus b_i$ is calculated and now we have to obtain $a_i \oplus c_{i-1}$ ($a_i$ have new value after step 1).

Observe the carry equation of full adder, *i.e.*,

$$c_i = a_i.b_i + (a_i \oplus b_i)c_{i-1}$$

The term $a_i.b_i$ generates a carry in the $i^{th}$ position, while $(a_i \oplus b_i)c_{in}$ term merely propagates the incoming carry to the next stage if $(a_i \oplus b_i) = 1$.

Since $a_i$ (new) actually represents $a_i \oplus b_i$ (for old $a_i$, $b_i$), the carry propagation simply depends on $a_i$, *i.e.*, if $a_i$ (new) = 1, the incoming carry is also given to the next stage. On the other hand a stage generates a carry if old values of $a_i$ and $b_i$ (before step 1) were both 1. This means new value of $a_i$ is 0 ($a_i \leftarrow a_i \oplus b_i$ ; *i.e.*, $a_i = 1 \oplus 1 = 0$). Thus a stage generates a carry if $a_i$ is '0' and $b_i$ is '1'.

The second term in carry equation, *i.e.*, $(a_i \oplus b_i)c_{in}$ is equivalent $a_i$ (new). $c_{in}$. Thus carry is propagated if $a_i$ is 1 (after step 1).

In short the step 1 and step 2 functions are :

Step 1 : (Pulse $P_1$) : Toggle the bit $a_i$ of A register if the bit $b_i$ of B register is 1 (for all bits).

Step 2 : (Pulse $P_2$) : Toggle the bit $a_i$ of A register if $a_{i-1}$ is 1 and there is an incoming carry or $a_{i-1}$ is 0 and $b_{i-1}$ is 1 (for all bits).

**Example 5.6** : Illustrate the 2-step addition with the following unsigned numbers.

(*a*)
A = 0 1 0 1 1 0 0
⊕ B = 0 0 1 0 1 1 0

Step 1 : A = 0 1 1 1 0 1 0 A ← A ⊕ B
⊕
Step 2 : carries 1 1 1 1 0 0 0

Result A = 1 0 0 0 0 1 0 A ← A ⊕ Carries

(*b*)
A = 1 1 0 1 1 0 0 1 1 1 0 1
⊕ B = 0 0 0 0 1 1 0 0 1 0 0 1

Step 1 : A = 1 1 0 1 0 1 0 1 0 1 0 0
⊕
Step 2 : carries 0 0 1 1 0 0 1 1 0 0 1 0

Result A = 1 1 1 0 0 1 1 0 0 1 1 0

## EXERCISES

1. Design a full subtractor (similar to full adder).

2. Using the full subtractors designed in (1), design the 4-bit adder/subtractor for numbers in sign-magnitude form.

3. Repeat (2) for numbers in sign-2's complement form.

4. Illustrate the 2-step addition method for the following in sign-magnitude and sign 2's complement form.

(*i*) (+13) + (+7)
(*ii*) (+13) + (−7)
(*iii*) (+15) + (+10)
(*iv*) (−15) + (−10)
(*v*) (−15) − (−10)

Use 5-bit adder (including sign bit). Show also the overflow flag value.

5. The $i^{th}$ bit of an operand may be 0 or 1 with equal probability, compute the following :

(*i*) Probability of $i^{th}$ stage of an adder generating a carry.

(*ii*) Probability of $i^{th}$ stage propagating an incoming carry.

(*iii*) Probability of a 4-bit adder propagating an incoming carry.

**6.** Design a 4-bit subtractor using a 2-step method (similar to the 2-step adders discussed).

**7.** It is required to know the number N where the N=number of 1's contained in a 12-bit register. Using only 4-bit adders, design the required circuit which accepts 12 bit input and produces a binary number N which indicates the number of 1's in the register.

**8.** (*a*) Formulate a block diagram of an adder stage (similar to full adder) which can be used for parallel addition of three *n*-bit binary numbers.

(*b*) Work out the detailed logic design of the above adder stage.

(*c*) Give the cascade connections of the above blocks to form an *n*-bit adder for adding three binary numbers.

**9.** Design a 4-bit adder using 2-step addition method.

**10.** Derive the various possible logic expressions for setting the overflow in sign-2's complement addition/subtraction.

**11.** (*a*) Give the sign-2's complement representation for $x=+13)_{10}$ and $y=-17)_{10}$, assume 6 bits (including sign bit).

(*b*) Show step by step, how the following operations shall be performed.

(*i*) $x+y$

(*ii*) $x-y$

(*iii*) $y-x$

**12.** Repeat (11) using 7 bits (including sign).

**13.** For the 2-step addition method discussed, design the complete logic for 4-bit adder for numbers in sign-complement form.

**14.** Using 74181 and 74182 blocks, design the carry lookahead adders for adding two :

(*a*) 100-bit numbers

(*b*) 40-bit numbers

(*c*) 32-bit numbers

(Give proper logical values to unused inputs of 74181*s* and 74182*s*).

**15.** Assuming the time delays of 20 *ns* for 74181 and 12*ns* for 74182, determine the addition times for the adders designed in (14).

□

# 6

# DECIMAL ARITHMETIC

DIGITAL computer programs work on various numerical and non-numerical data. The character sets (alphabets) of various programming languages like Fortran, Algol, PL/I etc., consist of 26 English alphabets, 10 numeric digits and some special symbols like +, ., , etc. Such information is represented in computers by means of codes.

## 6.1. CODES

Digital systems are built using binary devices, hence ultimately the information within a system is always in some binary form. Coding is a technique by which a set of objects is represented in a machine by a set of binary numbers called codes. As an example, consider a machine which operates on the following set :

$$S=\{A, B, C, F, G\}$$

This set of letters could be represented by a set $S_N$ of binary numbers shown below and the mapping $S \rightarrow S_N$ shown in Table 6.1.

$$S_N=\{000, 001, 010, 011, 100\}.$$

In this example, we have represented the five objects A, B, C, F and G by five 3-bit binary numbers. There are numerous ways of mapping S onto $S_N$ giving rise to various

TABLE 6.1

| S | $S_N$ |
|---|---|
| A | 000 |
| B | 001 |
| C | 010 |
| F | 011 |
| G | 100 |

codes, and Table 6.1 shows one of the codes The number of bits required to code a set of $n$ objects is given by :

$$k=[\log_2 n],$$

where $[x]$ denotes smallest integer $\geqslant x$ and $k$=number of bits required in the code.

### 6.1.1. Decimal Numbers and Their Codes

In the last section, we have discussed how the information can be represented by binary codes. When a machine has to deal with decimal numeric data in arithmetic unit, codes of four bit length are usually used (because the total number of decimal digits are ten only). Decimal numbers represented

in this manner are called Binary Coded Decimal (BCD) numbers, and the codes are called BCD codes. Table 6.2 lists some of the important BCD codes.

TABLE 6 2. **Some Important BCD Codes**

| *digit* | *8-4-2-1* W X Y Z | Excess 3 | 2-4-2_1 | 5-2-1-1 |
|---|---|---|---|---|
| 0 | 0 0 0 0 | 0 0 1 1 | 0 0 0 0 | 0 0 0 0 |
| 1 | 0 0 0 1 | 0 1 0 0 | 0 0 0 1 | 0 0 0 1 |
| 2 | 0 0 1 0 | 0 1 0 1 | 0 0 1 0 | 0 0 1 0 |
| 3 | 0 0 1 1 | 0 1 1 0 | 0 0 1 1 | 0 1 0 1 |
| 4 | 0 1 0 0 | 0 1 1 1 | 0 1 0 0 | 0 1 1 1 |
| 5 | 0 1 0 1 | 1 0 0 0 | 1 0 1 1 | 1 0 0 0 |
| 6 | 0 1 1 0 | 1 0 0 1 | 1 1 0 0 | 1 0 1 0 |
| 7 | 0 1 1 1 | 1 0 1 0 | 1 1 0 1 | 1 0 1 1 |
| 8 | 1 0 0 0 | 1 0 1 1 | 1 1 1 0 | 1 1 1 0 |
| 9 | 1 0 0 1 | 1 1 0 0 | 1 1 1 1 | 1 1 1 1 |

All these codes (except excess 3) are positional in the sense that each bit has a weightage, and the weighted sum of the bits equals the digit represented by a code number. Excess 3 code is not positional, but it is obtained from 8-4-2-1 code by adding 3 (hence the name excess 3).

The codes shown above have certain properties. All the codes except 8-4-2-1 code are self complementing *i.e.*, 9's complement of any digit could be obtained by simply complementing the individual bits, while 8-4-2-1 code is easily ameanable for addition of decimal numbers because these code words represent natural binary numbers for which arithmetic circuits could be easily constructed using binary adders.

A given decimal number is represented in the machine by individually coding its digits in the required code.

**Example 6.1.** Represent decimal numbers 158, and 276 in 8-4-2-1 and 5-2-1-1 codes.

Referring to the table 6.2, we write codes of digits of the numbers as shown below :

| Code | 1 | 5 | 8 | 2 | 7 | 6 |
|---|---|---|---|---|---|---|
| 8-4-2-1 | 0001 | 0101 | 1000 | 0010 | 0111 | 0110 |
| 5-2-1-1 | 0001 | 1000 | 1110 | 0100 | 1011 | 1010 |

The most commonly used code in digital computers is the 8-4-2-1 code. Hereafter the BCD numbers shall mean 8-4-2-1 BCD numbers, unless otherwise mentioned.

## 6.2. DECIMAL ADDITION OF UNSIGNED INTEGERS

In this section, the development of logic for the addition of decimal numbers is presented. Consider the numbers X=793 and Y=825 coded in the 8-4-2-1 code.

| | | | |
|---|---|---|---|
| X= 793 | X=0111 | 1001 | 0011 |
| +Y= 825 | +Y=1000 | 0010 | 0101 |
| 1100 | 0 0000 | 0000 | 1110←carries |
| S=1618 | S* 1111 | 1011 | 1000 |

S*=Sum obtained in the binary addition of X and Y.

If we add these coded numbers with the assumption that they are pure binary numbers, we shall obtain the sum as shown above. Note that only least significant digit of the sum has correct code, while other digit positions of sum have not only invalid codes, but also decimal carries are missing. The sum S* obviously is not correct, but could be corrected by adding a correction number 660 as shown below :

| | | | | |
|---|---|---|---|---|
| | S*=1111 | 1011 | 1000 |
| | correction→0110 | 0110 | 0000 |
| | 1 1 | | ←carries |
| correct | S= 1 0110 | 0001 | 1000 |
| decimal sum | S= 1 6 | 1 | 8 |

We can formulate the rules regarding correction of an incorrect sum digit to a correct value as follows. This discussion shall pave the way for obtaining the logic design of a single stage decimal adder (decimal full adder). The decimal full adder adds two decimal digits and an incoming carry ; and produces a decimal sum and a decimal carry as outputs. Since digits take values 0 to 9, while carry varies from 0 to 1 only, the minimum and maximum values of the sum of these three inputs are 0 and 19 respectively. Therefore, correct sum will be a 5-bit output as a result of binary addition of these three

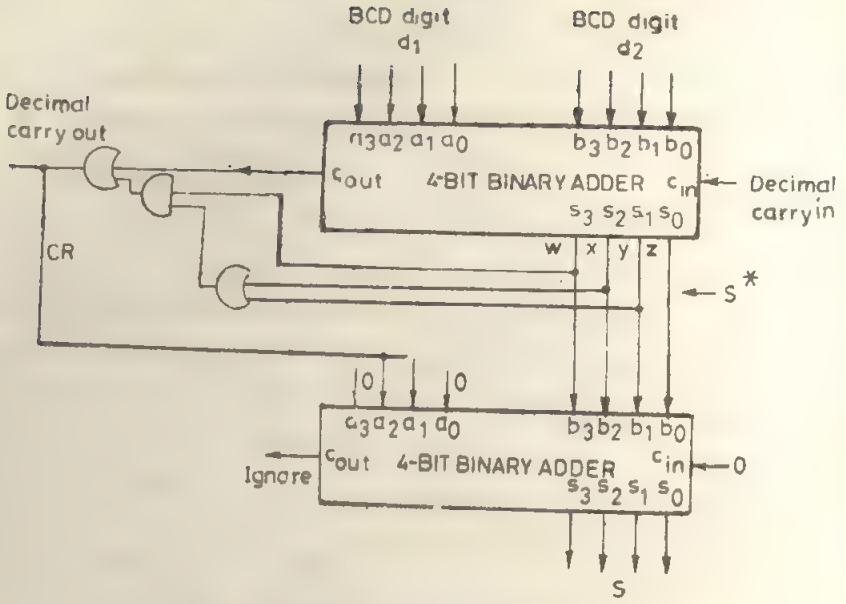



FIG. 6.1 DECIMAL FULL ADDER (DFA)

inputs. Table 6.3 gives the incorrect sum, correct sum and the correction.

From the Table 6.3 it may be observed that if the sum is greater than 9, we have to apply a correction, otherwise no correction is required. These combinations of incorrect sum could be recognised by a simple combinational logic circuit having inputs as bits of incorrect sum, and output signalling a correction To carry out the correction, we use another 4-bit adder, whose one of the operands is the incorrect sum. The other operand shall be either 6 or 0 depending upon whether a correction is required or not. Looking at the Table 6.3 it can be seen that a correction is required when either '$c$' is true or $w.(x+y)$ is true. The complete decimal adder for a stage, can now be constructed as shown in Fig. 6.1. Note that whenever there is a correction, decimal carry out is also present, thus CR signal itself could be given as decimal carry out.

Decimal addition can also be performed by another way as follows. In the method to be discussed, one of the operands is biased by 6 in individual digits, and the binary addition is performed. The correction table is shown in Table 6.4. Here the correction is required if there exists no carry in the binary addition from a 4-bit group, otherwise no correction is required. The required correction is substraction of 6 from individual incorrect sum digits.

**Example 6.2.** Illustrate addition of 267 and 548.

| | | | |
|---|---|---|---|
| A | 0010 | 0110 | 0111 |
| | 0110 | 0110 | 0110 |
| Biased A | 1000 | 1100 | 1101 |
| +B | 0101 | 0100 | 1000 |
| | 0 1 | | 1←carries |
| | 1110 | 0001 | 0101 |
| correction | −0110 | 0000 | 0000 |
| | 1000 | 0001 | 0101 |
| | 8 | 1 | 5 |

The complete design of decimal full adder using this principle is shown in Fig. 6.2.

TABLE 6.3. **Correction Table for BCD Addition**

| *Decimal value of sum* | *Incorrect sum S* (Result of Binary Addition of $d_1$, $d_2$ and $c_{in}$)* | | *Correction* | *Correct sum and Decimal Carry-out* | |
|---|---|---|---|---|---|
| | C | 4 bit sum $w\ x\ y\ z$ | | Carry | Sum |
| 0 | 0 | 0 0 0 0 | Sum is correct upto 9 hence no correction (or add '0') | 0 | 0000 |
| 1 | 0 | 0 0 0 1 | | 0 | 0001 |
| 2 | 0 | 0 0 1 0 | | 0 | 0010 |
| 3 | 0 | 0 0 1 1 | | 0 | 0011 |
| 4 | 0 | 0 1 0 0 | | 0 | 0100 |
| 5 | 0 | 0 1 0 1 | | 0 | 0101 |
| 6 | 0 | 0 1 1 0 | | 0 | 0110 |
| 7 | 0 | 0 1 1 1 | | 0 | 0111 |
| 8 | 0 | 1 0 0 0 | | 0 | 1000 |
| 9 | 0 | 1 0 0 1 | | 0 | 1001 |
| 10 | 0 | 1 0 1 0 | Sum is incorrect but by adding six (0110) we get the correct sum | 1 | 0000 |
| 11 | 0 | 1 0 1 1 | | 1 | 0001 |
| 12 | 0 | 1 1 0 0 | | 1 | 0010 |
| 13 | 0 | 1 1 0 1 | | 1 | 0011 |
| 14 | 0 | 1 1 1 0 | | 1 | 0100 |
| 15 | 0 | 1 1 1 1 | | 1 | 0101 |
| 16 | 1 | 0 0 0 0 | | 1 | 0110 |
| 17 | 1 | 0 0 0 1 | | 1 | 0111 |
| 18 | 1 | 0 0 1 0 | | 1 | 1000 |
| 19 | 1 | 0 0 1 1 | | 1 | 1001 |

**Illustration** : Incorrect sum 0 1100 is corrected by adding six

$$\begin{array}{r} 0\ 1100 \\ +0110 \\ \hline 10010\ (12) \end{array}$$

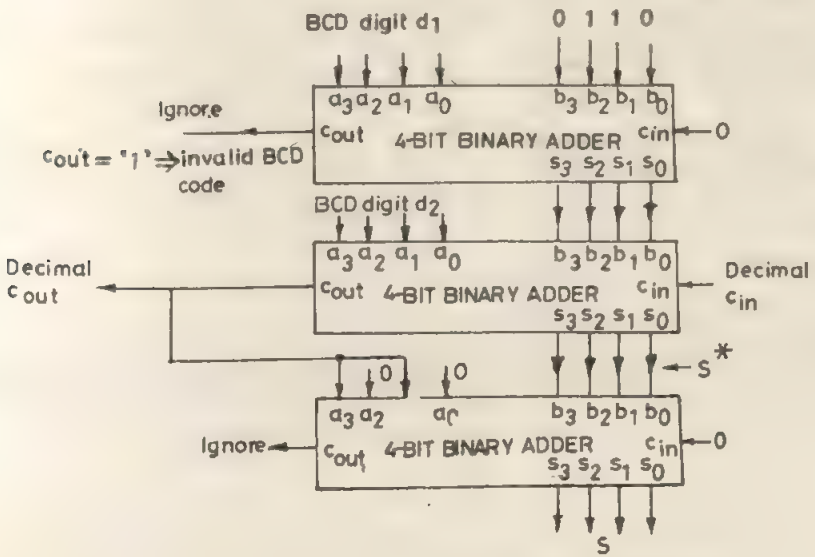



FIG.6·2 DECIMAL FULL ADDER USING BIASED OPERAND

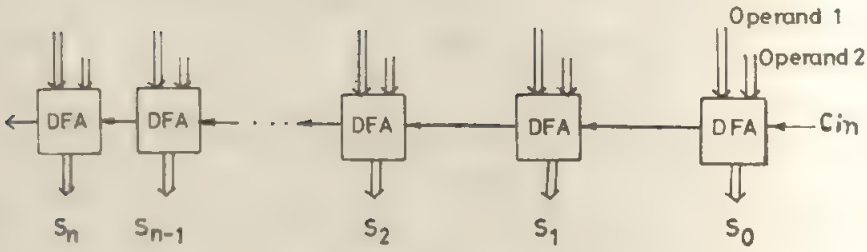



FIG.63 n-DIGIT PARALLEL DECIMAL ADDER

### 6.2.1. Comparison of the Two Methods

A decimal stage adder requires two 4-bit binary adders, if it is constructed using the first method, but three 4-bit binary adders are required if it is constructed using the second method. Therefore, one may consider the second method of no use. This is true if we are building a parallel *n*-digit adder by cascading the *n* blocks of adder stages as shown in Fig. 6.3. This *n*-digit decimal adder thus would require $2n$ or $3n$ 4-bit binary adders if first or second method is used respectively. But the second method has a clear advantage if the decimal addition is to be implemented using the binary adder available in the ALU. This point can be best illustrated through an example. Consider a machine having 32-bit binary adder, built using 4-bit adder chips (7483), on which we want to add the decimal numbers A=63345219 and B=26654787.

Using the first method one would have executed the following steps :

**TABLE 6 4 : Decimal Addition Using Biased Operand**

| *Decimal Value of Sum* | *Incorrect Sum S* (Binary Addition) of* $d_1+d_2+6+c_{in}$ C W X Y Z | *Correction* | *Correct Sum* |
|---|---|---|---|
| 0 | 0 0 1 1 0 | Subtract 0110 (or add 1010 ignoring the group carry) | 0 0000 |
| 1 | 0 0 1 1 1 | | 0 0001 |
| 2 | 0 1 0 0 0 | | 0 0010 |
| 3 | 0 1 0 0 1 | | 0 0011 |
| 4 | 0 1 0 1 0 | | 0 0100 |
| 5 | 0 1 0 1 1 | | 0 0101 |
| 6 | 0 1 1 0 0 | | 0 0110 |
| 7 | 0 1 1 0 1 | | 0 0111 |
| 8 | 0 1 1 1 0 | | 0 1000 |
| 9 | 0 1 1 1 1 | | 0 1001 |
| 10 | 1 0 0 0 0 | No correction | 1 0000 |
| 11 | 1 0 0 0 1 | | 1 0001 |
| 12 | 1 0 0 1 0 | | 1 0010 |
| 13 | 1 0 0 1 1 | | 1 0011 |
| 14 | 1 0 1 0 0 | | 1 0100 |
| 15 | 1 0 1 0 1 | | 1 0101 |
| 16 | 1 0 1 1 0 | | 1 0110 |
| 17 | 1 0 1 1 1 | | 1 0111 |
| 18 | 1 1 0 0 0 | | 1 1000 |
| 19 | 1 1 0 0 1 | | 1 1001 |

**Note :** Correction logic is derived from the state of C *i.e.*, if C=0 subtract 0110, if C=1 no correction (or subtract 0000).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | A=0110 | 0011 | 0011 | 0100 | 0101 | 0010 | 0001 | 1001 |
| Step 1 : Binary Addition | + B=0010 | 0110 | 0110 | 0101 | 0100 | 0111 | 1000 | 0111 |
| | | | | | | | 1 | |
| | 1000 | 1001 | 1001 | 1001 | 1001 | 1001 | 1010 | 0000 |
| Step 2 : Correction | + 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0110 | 0110 |
| | | | | | | 1←carry | | |
| | 1000 | 1001 | 1001 | 1001 | 1001 | 1010 | 0000 | 0110 |
| Step 3 : Correction | + 0000 | 0000 | 0000 | 0000 | 0000 | 0110 | 0000 | 0000 |
| | | | | | 1←carry | | | |
| | 1000 | 1001 | 1001 | 1001 | 1010 | 0000 | 0000 | 0110 |
| Step 4 ; Correction | + 0000 | 0000 | 0000 | 0000 | 0110 | 0000 | 0000 | 0000 |
| | | | | 1←carry | | | | |
| | 1000 | 1001 | 1001 | 1010 | 0000 | 0000 | 0000 | 0110 |
| Step 5 : Correction | + 0000 | 0000 | 0000 | 0110 | 0000 | 0000 | 0000 | 0000 |
| | | | 1←carry | | | | | |
| | 1000 | 1001 | 1010 | 0000 | 0000 | 0000 | 0000 | 0110 |
| Step 6 : Correction | + 0000 | 0000 | 0110 | 0000 | 0000 | 0000 | 0000 | 0110 |
| | | 1←carry | | | | | | |
| | 1000 | 1010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0110 |
| Step 7 : Correction | + 0000 | 0110 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| | 1←carry | | | | | | | |
| Result : | 1001 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0110 |

Now we illustrate the addition using the second method.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | A=0110 | 0011 | 0011 | 0100 | 0101 | 0010 | 0001 | 1001 |
| Step 1 : Bias | 0110 | 0110 | 0110 | 0110 | 0110 | 0110 | 0110 | 0110 |
| Biased | A=1100 | 1001 | 1001 | 1010 | 1011 | 1000 | 0111 | 1111 |
| Step 2 : Binary Addition | 0010 | 0110 | 0110 | 0101 | 0100 | 0111 | 1000 | 0111 |
| Biased A+B | 0 1 | 1 | 1 | 1 | 1 | 1 | | 1←carries |
| | 1111 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0110 |
| Step 3 : Correction | +1010 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 |
| (add 1010 to the incorrect sum digits with group carries disabled) | 1001 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0110 |

Here correction is applied for all groups which does not generate carry in step 2.

In the above example, it can be seen that the biased operand method is clearly advantageous, since it required only 3 steps, while the first method required 7 steps. In general, the addition of $n$-digit decimal numbers using first method shall require $n$ steps (in the worst case), while the second method always requires only 3 steps.

## 6.3. ADDITION/SUBTRACTION OF SIGNED DECIMAL NUMBERS

Signed decimal numbers could be in any of the following three forms :

(1) Sign-Magnitude form

(2) Sign-10's complement form

(3) Sign-9's complement form.

The numbers in the above forms can be added using techniques of chapter 5 (see section 5.4). Only difference here is that instead of 1's or 2's complement we shall use 9's or 10's complement respectively and the full adder of binary addition is replaced by a BCD full adder. Therefore, we shall discuss how 9's or 10's complement functions could be implemented using logic circuits.

### 6.3.1. 9's Complement Circuits

A 9's complement of a decimal digit $d$ is defined as :

$$d'' = 9 - d \qquad \ldots(6.1)$$

We shall discuss now how to bulid a logic circuit implementing the above equation for 8-4-2-1 BCD code.

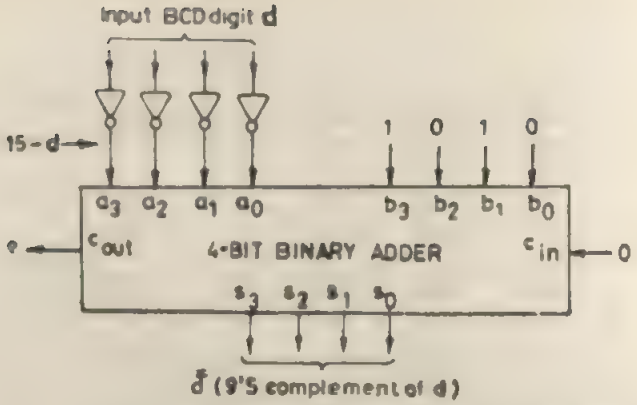



FIG. 6·4 9'S COMPLEMENTOR USING EQUATION (6·2)

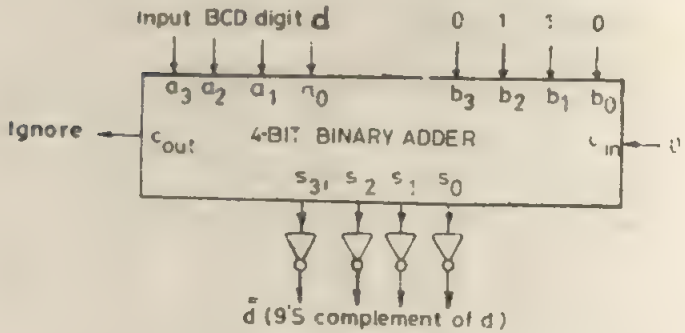



FIG 6·5 9'S COMPLEMENTOR USING EQUATION (6·3)

Equation (6.1) could be written as :

$$d'' = 15 - d - 6$$

or $$d'' = (15 - d) - 6 \quad \ldots(6.2)$$

or $$d'' = 15 - (d + 6) \quad \ldots(6.3)$$

The Equation (6.2) has two terms *viz.*, $15 - d$ and 6. $d$ is a 4-bit binary number (8-4-2-1 code), hence $15 - d$ represents its 1's complement. Therefore, if one wants to implement a 9's complement of a BCD digit it could be obtained by first complementing each bit of $d$ and then adding $-6$. The logic circuit for this is shown in Fig. 6.4.

Similarly using Equation (6·3), we obtain $d''$ as shown in Fig. 6.5.

Using any of these 9's complementing blocks developed, we can complement a given $n$ digit decimal number as shown in Fig. 6.6 or 6.7.

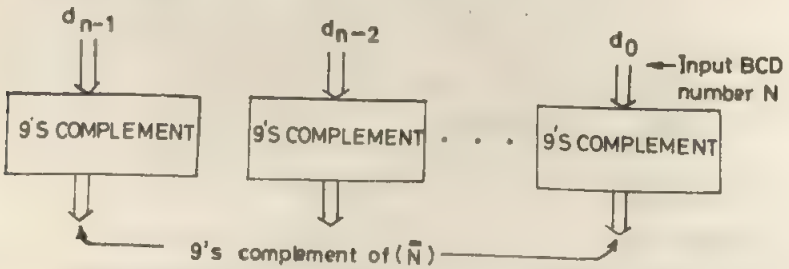



FIG. 6·6 PARALLEL 9'S COMPLEMENTOR FOR BCD NUMBERS

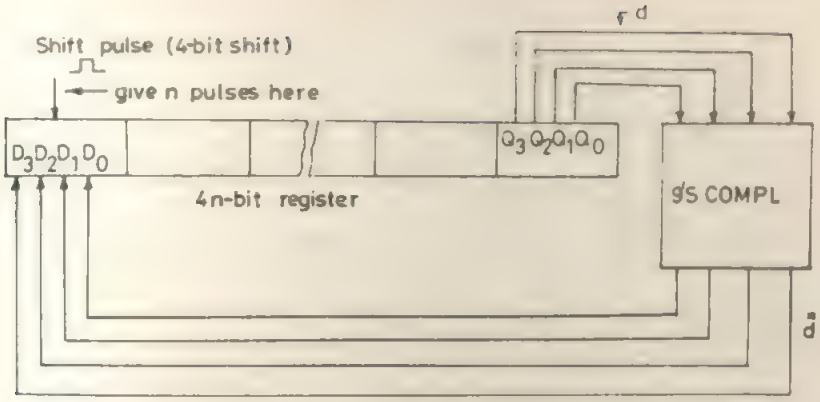



FIG. 6·7 SERIAL 9'S COMPLEMENTOR

### 6.3.2. 10's Complementing Circuits

10's complement of a BCD digit $d$ is given by :

$$d' = \begin{cases} 10-d & \text{if } d \neq 0 \\ 0 & \text{if } d = 0 \end{cases} \quad \ldots(6.4)$$

Thus for a non-zero $d$, we have

$$d' = 10-d = 15-5-d$$

*i.e.*, $d' = (15-d)-5$ ...(6.5)

or $d' = 15-(d+5)$ ...(6.6)

The 10's complement of non-zero digit is obtained as follows :

(*i*) *Usirg Equation* (6.5) :

1. Complement each bit
2. Add 1011 (−5 in 2's complement form)

(*ii*) *Using Equation* (6.6) :

1. Add 0101
2. Complement the bits of result of addition in 1.

Using the Equation (6.5) the complete logic diagram is built and is shown in Fig. 6.8. Here $d=0$, is decoded and the appropriate second operand for the 4-bit binary adder is generated as per the following discussion.

The input A to the 4-bit adder is $15-d$. If $d \neq 0$ then the input B should be $1011)_2$ otherwise it should be $0001)_2$ (If $d=0$, then A=1111, hence 0001 must be added to produce 0000 at the output). The line Z in Fig. 6.8 indicates whether $d=0$ or $d \neq 0$ by producing a logic '0' or '1' at Z. The operand B for these values of Z is shown in Table 6.5. This table defines switching functions $b_3$, $b_2$, $b_1$, $b_0$ with Z as input

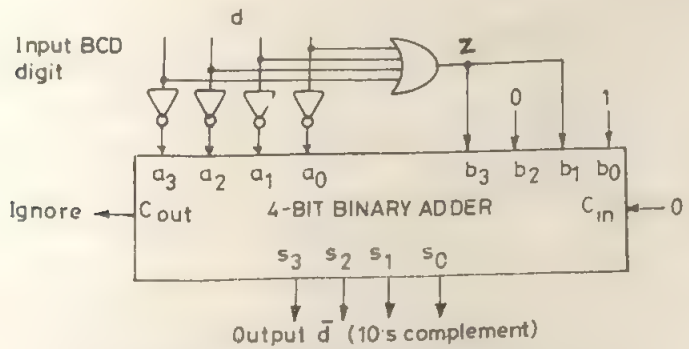



FIG.6·8 10'S COMPLEMENTOR USING EQUATION (6·5)

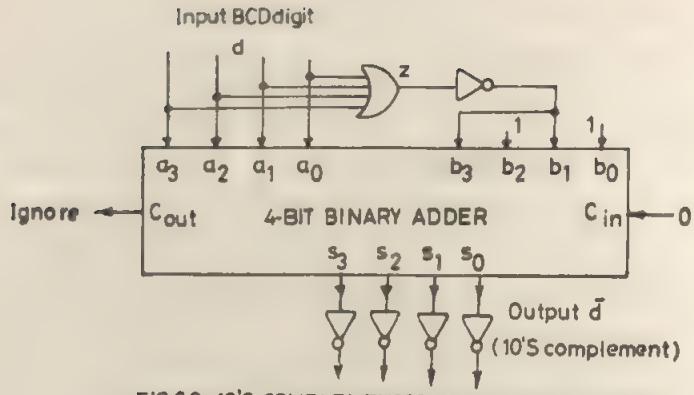



FIG.6-9 10'S COMPLEMENTOR USING EQUATION(6-6)

TABLE 6.5

| Z | Bits of the operand B $b_3$ $b_2$ $b_1$ $b_0$ | Comments |
|---|---|---|
| 0 | 0 0 0 1 | $d=0$ |
| 1 | 1 0 1 1 | $d\neq 0$ |

From the table we write ;

$b_3=Z$
$b_1=Z$
$b_2=0$
$b_0=1$

The operand B is formed from Z, using the above relations and the complete circuit is shown in Fig. 6.8. Alternate circuit designed using Equation (6.6) is shown in Fig. 6 9.

To implement a 10's complementer for an *n*-digit decimal number, we require a logic block doing both 9's and 10's complement functions under an external control signal. (See algorithm 2.5 of chapter 2). This block could be designed as follows :

Let TN be the control input which determines 10's or 9's complement function. Then, by noting TN and incoming digit *d*, we can generate second operand as required so that 4-bit adder will produce a required complement. Using Equations (6.2) and (6.5), this block is built and is shown in Fig. 6.10.

Logic equations for bits of B are developed as per the switching functions shown in Table 6.6.

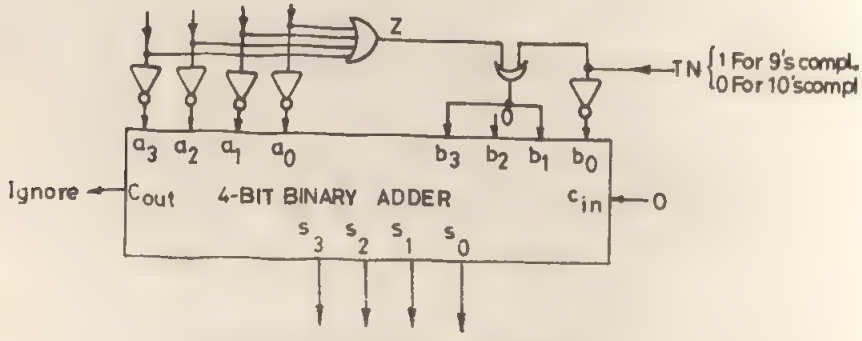



FIG.6-10 10'S/9'S COMPLEMENTOR

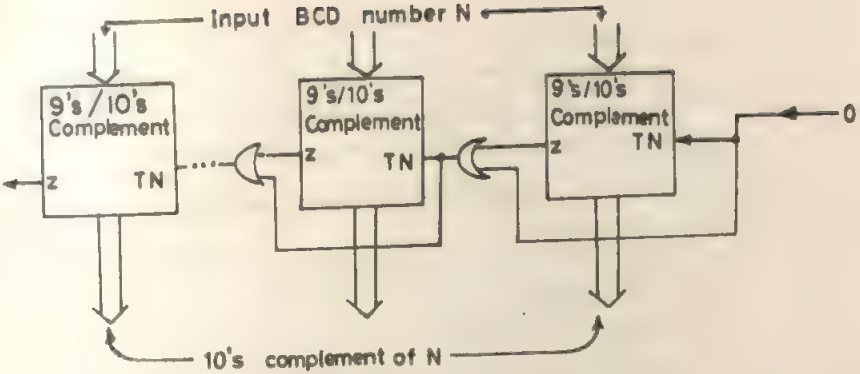



FIG. 6-11 PARALLEL 10'S COMPLEMENTOR

TABLE 6.6

| *Inputs* TN Z | *Out-puts* $b_3$ $b_2$ $b_1$ $b_0$ | *Comments* |
|---|---|---|
| 0 0 | 0 0 0 1 | 10's complement of $d=0$ is 0 |
| 0 1 | 1 0 1 1 | 10's complement of $d\neq 0$ is (10—d) |
| 1 0 | 1 0 1 0 | 9's complement of $d$ is $9-d$ |
| 1 1 | 1 0 1 0 | 9's complement of $d$ is $9-d$ |

Therefore, $b_3=TN+Z$
$b_2=0$
$b_1=TN+Z$
$b_0=\overline{TN}$

Using the block shown in Fig. 6 10 and the Algorithm 2.5 the 10's Complementor for $n$-digit numbers is shown in Fig. 6.11.

### 6.3.3. n-Digit BCD Adder/Subtractor for Numbers in Sign —10's Complement Form

Complete logic design of an $n$-digit BCD adder/subtractor for numbers in sign-10's complement form is shown in Fig. 6.12. In the adder shown, the add/sub (A/S) line controls the 2 to 1 multiplexor which selects B in true or 9's complement form depending upon the operation. Also the decimal carry-in of the $n$-digit decimal adder is connected to this line. The adder subtractor works on the principles of binary adder shown in Fig. 5.9.

## 6.4. NUMBER CONVERSION ALGORITHMS

The conversion of numbers from binary form to BCD and vice-versa is discussed in this section. We shall develop algorithms for the above conversions such that a digital logic or a program could be synthesised to implement the algorithms.

### 6.4.1. Binary To BCD Conversion

An algorithm to carry out binary to BCD conversion is discussed here. The algorithm requires two registers, one register initially holds the binary number to be converted. The result of the conversion appears in the other registers (BCD register) which is made '0' initially (for unsigned integer conversions). The given binary number N is put in the register BR (Binary Register). Assume that the radix point exists at the junction of BR and BCD registers. With this assumption the composite number contained in these registers is :

$$D+\frac{N}{2^n}$$

where, D is the decimal part of the composite number, (*i.e.*, contents of BCD register) which is initially zero, and $n$ is the number of

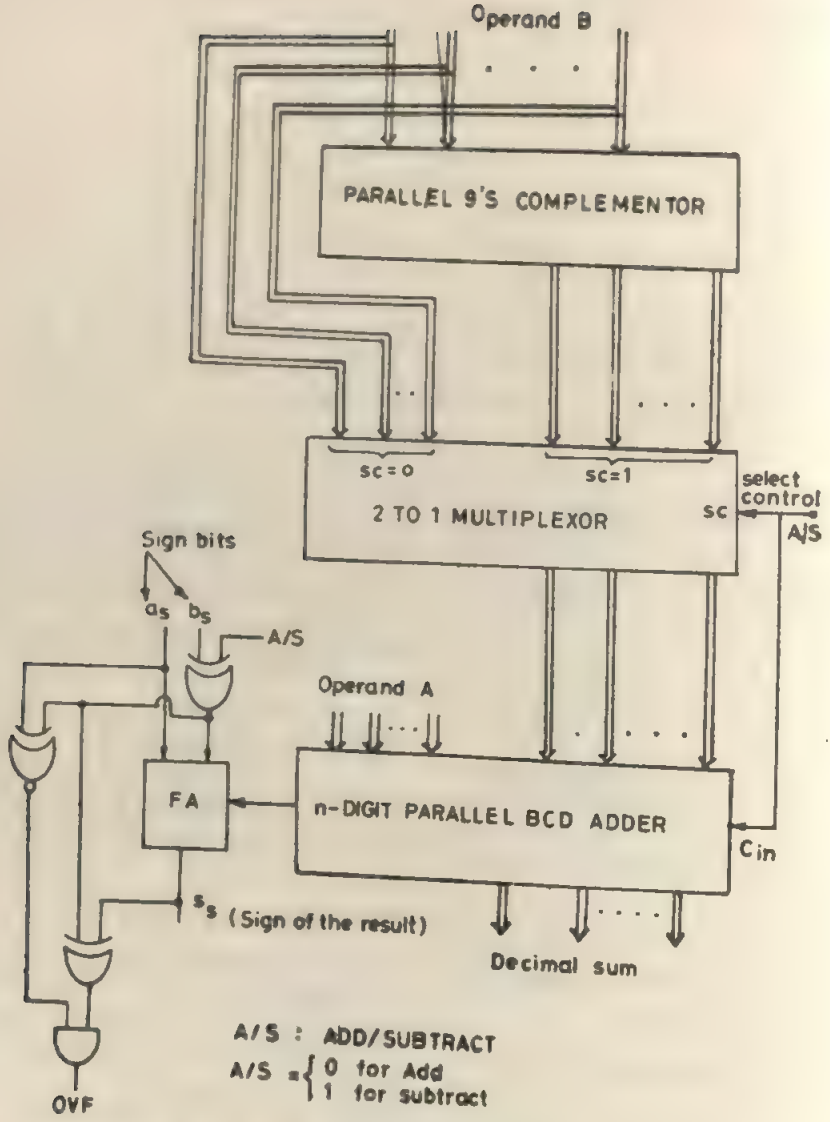



FIG 6-12 BCD ADDER FOR NUMBERS IN SIGN–10'S COMPLEMENT FORM

bits in the binary register. The actual number N is :

$$N=\left(D+\frac{N}{2^n}\right)\times 2^n \text{ (D is 0 initially)}$$

In effect we are multiplying the composite number by $2^n$. This is done in $n$ steps, each step multiplying the composite number by 2.

Multiplication by 2 on the binary part of the composite number is equivalent to shifting left the binary part by 1 position. On the other hand multiplication of D(BCD part) by 2 is equivalent to adding D to D itself. D is a BCD number hence this addition is equivalent to :

(*i*) Binary addition of D with D

(*ii*) Addition of 6 to the digits requiring correction.

These steps are equivalent to the following algorithm.

**Algorithm 6.1**

Do $n$ times :

(*i*) Shift left combined Binary and BCD register by 1 bit.

(*ii*) Add $0110)_2$ to a BCD digit if (*a*) it produced a carry in the above shift (equivalently a 1 crossing through the boundary of a 4-bit group) or (*b*) it is greater than $1001)_2$.

**Example 6.3** : Convert 10110011 into BCD.

| Steps | BCD Reg. $d_2$ | $d_1$ | $d_0$ | Binary Reg. |
|---|---|---|---|---|
| | 0000 | 0000 | 0000 | 1011 0011 |
| 1. (*a*) Shift left | 0000 | 0000 | 0001 | 0110 011b |
| (*b*) No correction | | | | |
| 2. (*a*) Shift left | 0000 | 0000 | 0010 | 1100 11bb |
| (*b*) No correction | | | | |
| 3. (*a*) Shift left | 0000 | 0000 | 0101 | 1001 1bbb |
| (*b*) No correction | | | | |
| 4. (*a*) Shift left | 0000 | 0000 | 1011 | 0011 bbbb |
| (*b*) Correction + | | | 0110 | |
| | 0000 | 0001 | 0001 | 0011 bbbb |
| 5. (*a*) Shift left | 0000 | 0010 | 0010 | 011b bbbb |
| (*b*) No correction | | | | |
| 6. (*a*) Shift left | 0000 | 0100 | 0100 | 11bb bbbb |
| (*b*) No correction | | | | |
| 7. (*a*) Shift left | 0000 | 1000 | 1001 | 1bbb bbbb |
| (*b*) No correction | | | | |
| 8. (*a*) Shift left | 0001 | 0001 | 0011 | bbbb bbbb |

(*b*) '1' Crossed from $d_0$ and $d_1$ : correction

*STOP* ($n$=8 steps)

| | | | | |
|---|---|---|---|---|
| | | 0110 | 0110 | |
| | 0001 | 0111 | 1001 | bbbb bbbb |
| Result : | 1 | 7 | 9 | Decimal |

**Note :** The b is written in a vacated position of binary register (the b will usually be 0).

While illustrating the algorithm, a correction is made (addition of 6) to any BCD digit which is greater than or equal to $(1010)_2$ or when a 1 crossed the group boundry, thus this involves decoding of both these conditions. This could be avoided if we check digits before shifting and correct them in advance. Any BCD digit $\geqslant 5$ is going to produce a correction after shifting (it would be $\geqslant$ ten). We thus apply a correction of $6/2=0011)_2$ if a digit (before shifting) exceeds or equals $0101)_2$.

Now we have a better algorithm as below :

**Algorithm 6.2 :**

Do n times

(*a*) if $d_i \geqslant 0101$ then $d_i=d_i+0011$ (for all $d_i$)

(*b*) shift left BCDR—BR by 1 bit position. (where $d_i$ is $i$th BCD digit)

**Example 6.4 :** Convert 10110011 into Binary.

| *Steps* | $d_2$ | $d_1$ | $d_0$ | |
|---|---|---|---|---|
| | 0000 | 0000 | 0000 | 1011 0011 |
| 1. (*a*) No correction | | | | |
| (*b*) Shift left | 0000 | 0000 | 0001 | 0110 011b |
| 2. (*a*) No correction | | | | |
| (*b*) Shift left | 0000 | 0000 | 0010 | 1100 11bb |
| 3. (*a*) No correction | | | | |
| (*b*) Shift left | 0000 | 0000 | 0101 | 1001 1bbb |
| 4. (*a*) Add 3 to $d_0$ | | | 0011 | |
| | 0000 | 0000 | 1000 | |
| (*b*) Shift left | 0000 | 0001 | 0001 | 0011 bbbb |
| 5. (*a*) No correction | | | | |
| (*b*) Shift left | 0000 | 0010 | 0010 | 011b bbbb |

| | | | | |
|---|---|---|---|---|
| 6. (*a*) No correction | | 0000 | 0100 | 0100 | 11bb bbbb |
| (*b*) Shift left | | | | | |
| 7. (*a*) No correction | | 0000 | 1000 | 1001 | 1bbb bbbb |
| (*b*) Shift left | | | | | |
| 8. (*a*) Add 3 to $d_0$, $d_1$ | + | | 0011 | 0011 | |
| | | 0000 | 1011 | 1100 | 1bbb bbbb |
| (*b*) Shift left | | 0001 | 0111 | 1001 | bbbb bbbb |
| STOP | Result : | | 1 | 7 | 9 |

By considering only magnitudes these algorithms could be used for converting signed numbers in sign-magnitude form. The same algorithms with different initial condition in BCD register can be made to work for binary numbers in sign -2's complement form. The converted number will be a BCD number in sign -10's complement form. The sign bit of a binary number in sign -2 s complement form has a negative weight (see Chapter 2). Thus a -1 is actually to be put initially in the BCD register if the number is negative. This is illustrated through the example 6.5.

**Example 6.5 :** Convert a signed binary number 1,0110011 in sign -2 s complement form into a signed BCD number in sign -10's complement form.

| | | Sign of BCD No. | $d_2$ 0000 | $d_1$ 0000 | $d_0$ 0000 1 ↓ | ↓ sign bit |
|---|---|---|---|---|---|---|
| No. | *Steps* | | | | | |
| | | 1 | 1001 | 1001 | 1001 | 10110011 |
| 1. (*a*) Correct $d_0$, $d_1$, $d_2$ | + | | 0011 | 0011 | 0011 | |
| | | 1 | 1100 | 1100 | 1100 | |
| (*b*) Shift left | | 1 | 1001 | 1001 | 1001 | 0110011b |
| 2. (*a*) Correct $d_0$, $d_1$, $d_2$ | + | | 0011 | 0011 | 0011 | |
| | | 1 | 1100 | 1100 | 1100 | |
| (*b*) Shift left | | 1 | 1001 | 1001 | 1000 | 110011bb |

| | | | | | |
|---|---|---|---|---|---|
| 3. (*a*) Correct $d_0$, $d_1$, $d_2$ | + | | 0011 0011 0011 | |
| | | 1 | 1100 1100 1011 | |
| (*b*) Shift left | | 1 | 1001 1001 0111 | 10011bbb |
| 4. (*a*) Correct $d_0$, $d_1$, $d_2$ | + | | 0011 0011 0011 | |
| | | 1 | 1100 1100 1010 | |
| (*b*) Shift left | | 1 | 1001 1001 0101 | 0011bbbb |
| 5. (*a*) Correct $d_0$, $d_1$, $d_2$ | + | | 0011 0011 0011 | |
| | | 1 | 1100 1100 1000 | |
| (*b*) Shift left | | 1 | 1001 1001 0000 | 011bbbbb |
| 6. (*a*) Correct $d_2$, $d_1$ | + | | 0011 0011 | |
| | | 1 | 1100 1100 0000 | |
| (*b*) Shift left | | 1 | 1001 1000 0000 | 11bbbbbb |
| 7. (*a*) Correct $d_1$, $d_2$ | + | | 0011 1011 | |
| | | 1 | 1100 1011 0000 | |
| (*b*) Shift left | | 1 | 1001 0110 0001 | 1bbbbbbb |
| 8. (*a*) Correct $d_1$, $d_2$ | + | | 0011 0011 | |
| | | 1 | 1100 1001 0001 | |
| (*b*) Shift left | | 1 | 1001 0010 0011 | bbbbbbbb |
| Result : | | 1 | 9 2 3 | |

in 10's complement

Note: Actual number is $-1\cdot10^3+923=-77$

Binary number given was $-1.2^7+2^5+2^4+2^1+1$

$=-128+51=-77$

### Logic Circuits For Algorithm 6.2

The algorithm 6.2 could be implemented by sequential or combinational circuits as discussed in the following paragraphs.

### Sequencial Logic Implementation

The logic circuit of Fig. 6.13 implements one step of the algorithm 6.2 by applying one pulse, as shown. The BCD register 4-bit groups are compared with 0100 and a 4-bit group having larger value than 0100 is replaced by corrected value (3 is added to it).

Let $w, x, y, z$ be the 4-bits of a group, then the correction logic is given (see table 6.7)

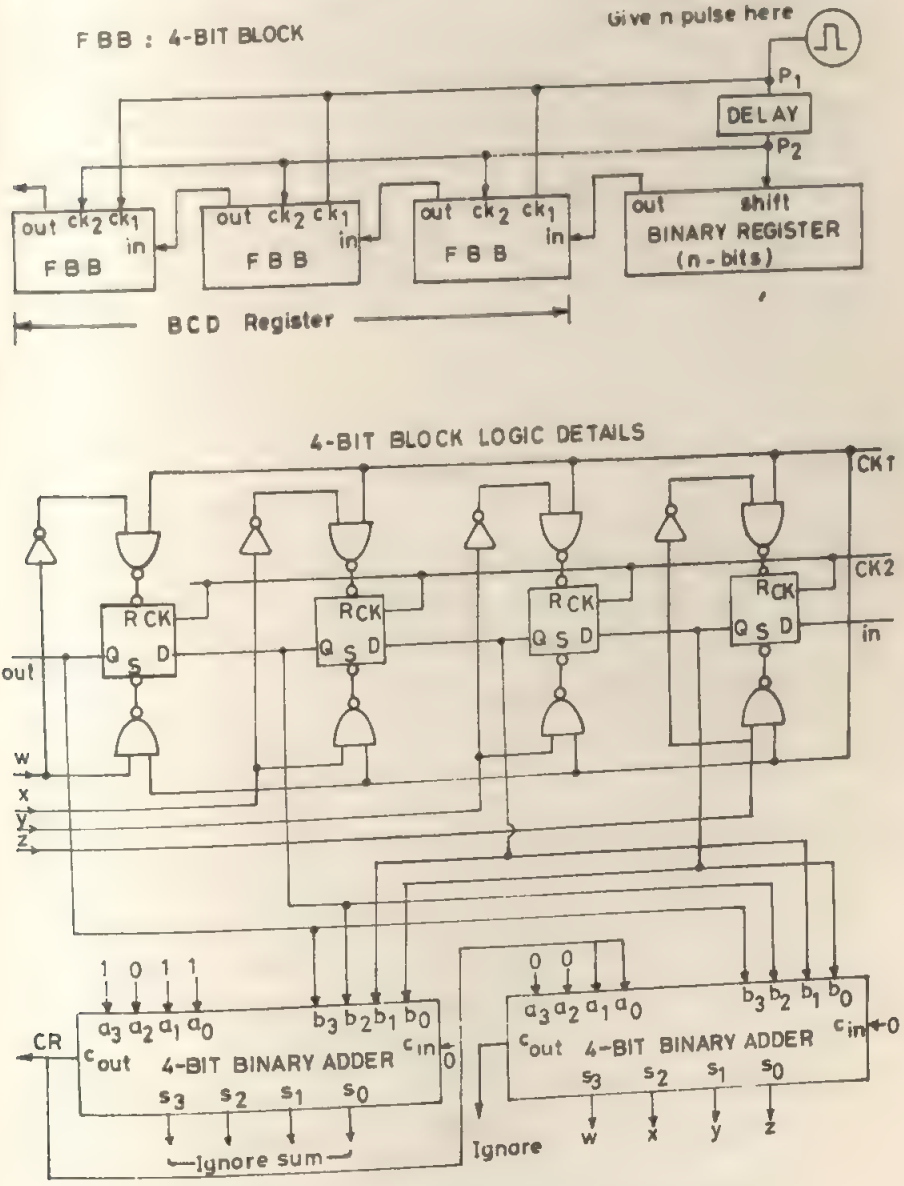



FIG.6-13 SEQUENTIAL LOGIC FOR BINARY TO BCD CONVERSION

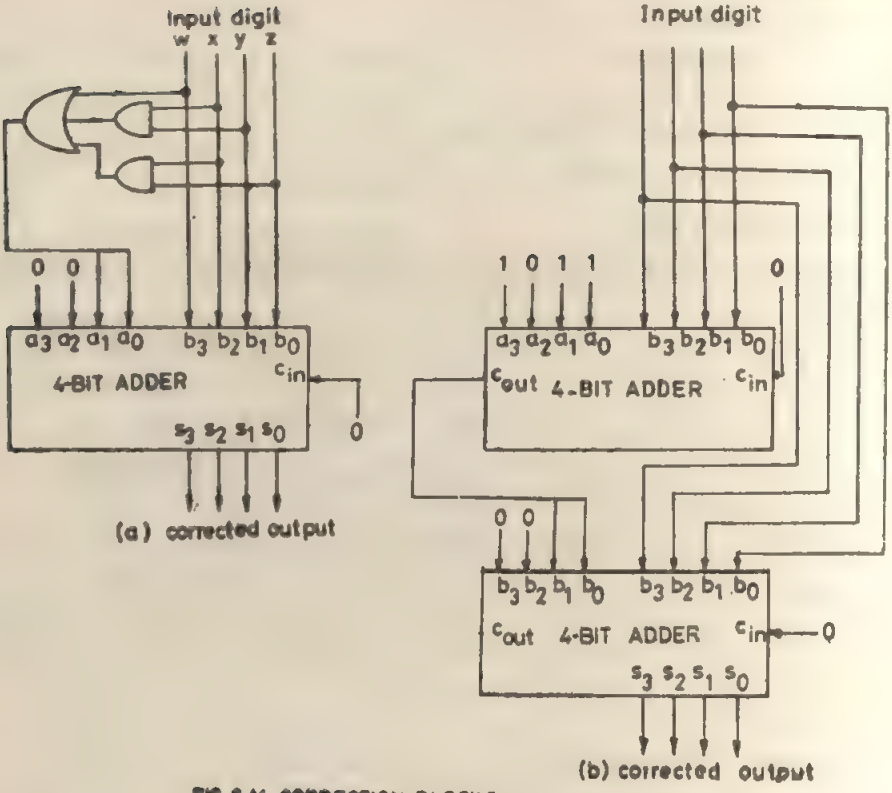



FIG. 6·14 CORRECTION BLOCKS

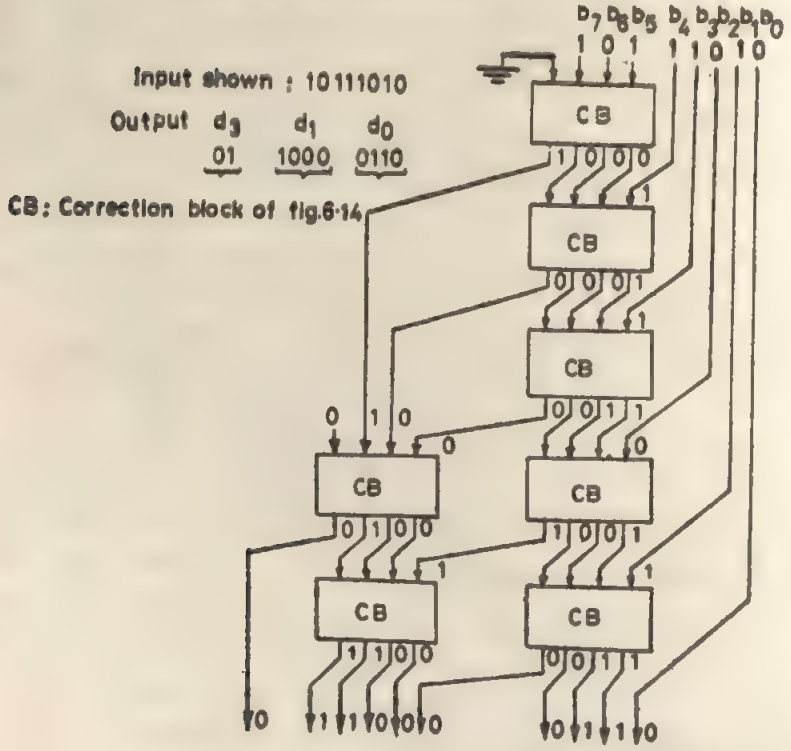



FIG. 6·15 PARALLEL BINARY TO BCD CONVERTER

by $w+x(y+z)$.

The blocks of Fig. 6.14 can be used in the iterative way to obtain the logic circuit of Fig. 6.15. Here, no clock is required. Binary input is applied and after a delay of propagation the BCD O/P is available. The circuit shown is for 8-bits binary numbers and could be extended for binary numbers with more bits.

### 6.4.2 BCD To Binary Conversion

BCD to binary conversion is a reverse process to that discussed for binary to BCD conversion. Here, the number in the BCD register is divided by $2^n$ in n steps, each step dividing the BCD register by 2. This will bring out remainder (0 or 1) into binary register. Remainder is decided by the least significant bit of a BCD number. The new contents of the BCD register will be quotient of the division. If we shift BCD register by 1 bit right, the binary number represented by bits of BCD register shall get divided by 2. But in BCD a 1 crossing between 4-bit groups represents a power of 10 instead of 16 and hence a crossed 1 must have the value equivalent to $10/2=0101)_2$ instead of $16/2=1000)_2$. From this, we formulate the following conversion algorithm.

**Algorithm 6.4** : BCD to Binary Conversion of Unsigned Integers

DO n times.

(*i*) Shift combined BCD-BR registers right by 1 bit.

(*ii*) If $d_i \geqslant 1000)_2$ then $d_i=d_i-0011)_2$ ($d_i$=BCD register 4 bit group, i=1, 2, 3,...k).

where n=Number of bits in the binary register k=Number of digits in BCD register.

**Example 6.6** : Convert 237 into Binary

| Steps | BCD Reg. $d_2$ | $d_1$ | $d_0$ | BR |
|---|---|---|---|---|
| | 0010 | 0011 | 0111 | |
| 1. (*a*) Shift right | 0001 | 0001 | 1011 | 1 |
| (*b*) $d_0 \geqslant 1000$ : correction to $d_0$ | | | −0011 | |
| | 0001 | 0001 | 1000 | 1 |
| 2. (*a*) Shift right | 0000 | 1000 | 1100 | 01 |
| (*b*) $d_1, d_0 \geqslant 100$ : correction to $d_1, d_0$ | | −0011 | −0011 | |
| | 0000 | 0101 | 1001 | |
| 3. (*a*) Shift right | 0000 | 0010 | 1100 | 101 |
| (*b*) $d_0 \geqslant 1000$ : correction to $d_0$ | | | −0011 | |
| | 0000 | 0010 | 1001 | |

| | | |
|---|---|---|
| 4. (*a*) Shift right | 0000 0001 0100 | 1101 |
| (*b*) No correction | | |
| 5. (*a*) Shift right | 0000 0000 1010 | 01101 |
| (*b*) $d_0 \geqslant 1000$ : | −0011 | |
| correction to $d_0$ | 0000 0000 0111 | |
| 6. (*a*) Shift right | 0000 0000 0011 | 101101 |
| (*b*) No correction | | |
| 7. (*a*) Shift right | 0000 0000 0001 | 1101101 |
| (*b*) No correction | | |
| 8. (*a*) Shift right | 0000 0000 0000 | 11101101 |
| (*b*) No correction | | |
| Result : | N | =11101101 |

**Implementation Of Algorithm 6.4 By Logic Circuits**

Sequential and combinational circuit implementations of the BCD to binary conversion algorithm are shown in Fig. 6.16 and 6.17 respectively. The BCD digits of shifted BCD register are compared for a 1 in MSB, and 0011 is substracted if a 1 is found in MSB. In Fig. 6.16 detailed correction logic for a BCD digit is shown. Similar circuits are put for all other BCD digits. For combinational logic implementation a basic block (CB) carrying out the correction is shown in Fig. 6.17 (a). This block is used in the iterative array of logic shown in Fig. 6.17 (b).

The algorithm 6.4 could also be used (with slight modification) for converting BCD numbers in sign -10's complement form into binary numbers in sign -2's complement form, with a difference in the shift operation. Here, we shift the BCD register using the arithmetic shift *i.e.*, sign bit of the BCD register is copied in the vacated bit position and the sign bit is left unaltered. We illustrate this by the following example.

**Example 6.7 :** Convert 1,53 (in sign -10's complement form) in to binary number in sign -2's complement form.

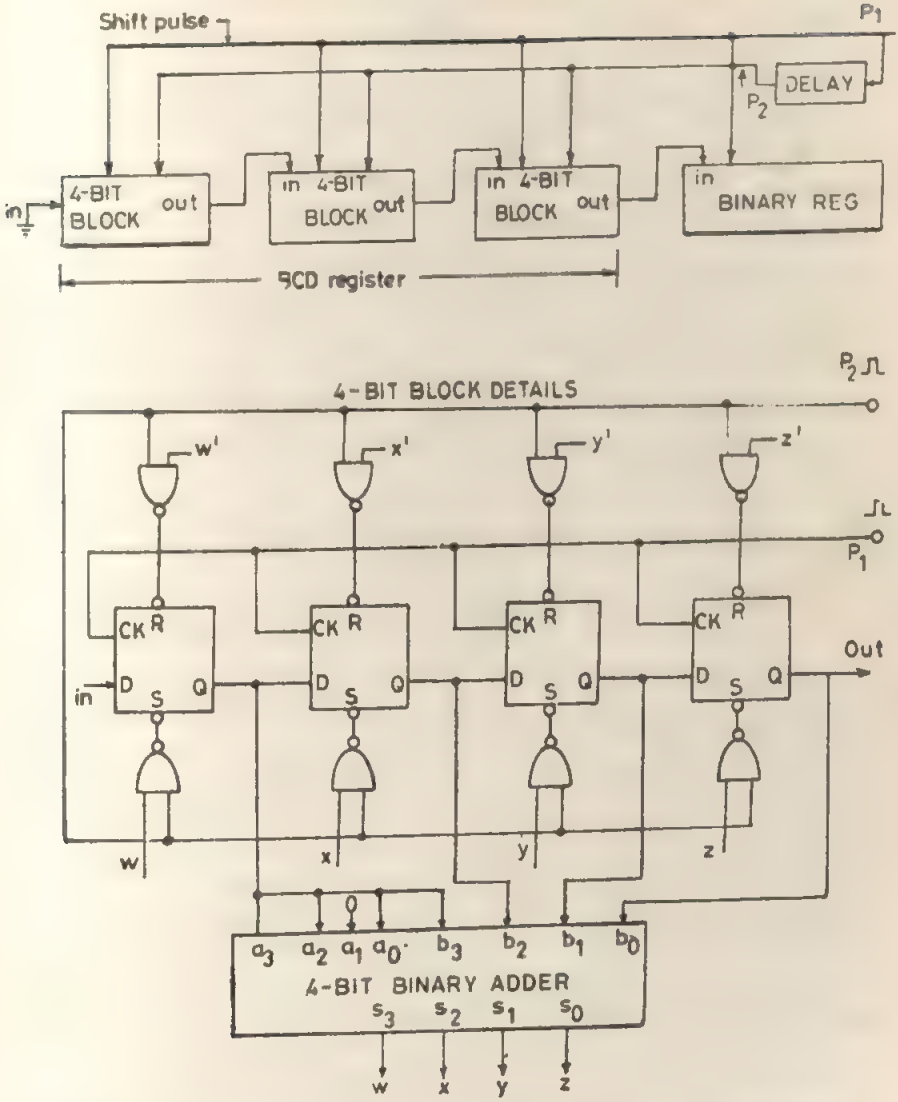



FIG. 6.16 SEQUENTIAL LOGIC IMPLEMENTATION OF ALGORITHM 6·4

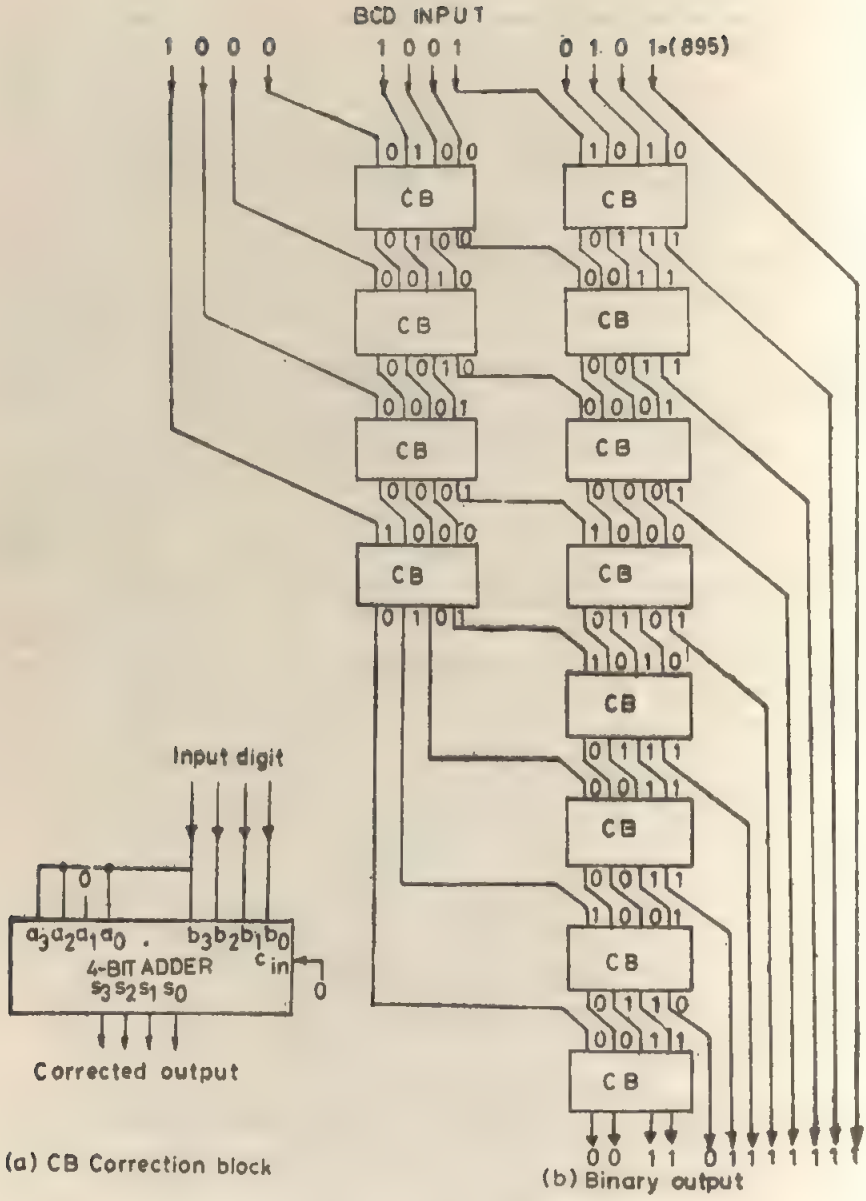



FIG. 6·17 BCD TO BINARY CONVERTOR

| *Steps* | Sign bit | BCD Register | Binary Register |
|---|---|---|---|
| | 1 | 0101 0011 | bbbbbbbb |
| 1. (*a*) Shift right | 1 | 1010 1001 | 1bbbbbbb |
| (*b*) Correction | | — 0011 0011 | |
| | 1 | 0111 0110 | 1bbbbbbb |
| 2. (*a*) Shift right | 1 | 1011 1011 | 01bbbbbb |
| (*b*) Correction | | — 0011 0011 | |
| | 1 | 1000 1000 | 01bbbbbb |
| 3. (*a*) Shift right | 1 | 1100 0100 | 001bbbbb |
| | | — 0011 | |
| (*b*) Correction | 1 | 1001 0100 | 001bbbbb |
| 4. (*a*) Shift right | 1 | 1100 1010 | 0001bbbb |
| | | — 0011 0011 | |
| (*b*) Correction | 1 | 1001 0111 | 0001bbbb |
| 5. (*a*) Shift right | 1 | 1100 1011 | 10001bbb |
| | | — 0011 0011 | |
| (*b*) Correction | 1 | 1001 1000 | 10001bbb |
| 6. (*a*) Shift right | 1 | 1100 1100 | 010001bb |
| | | — 0011 0011 | |
| (*b*) Correction | 1 | 1001 1001 | 010001bb |

| | | | |
|---|---|---|---|
| 7. (*a*) Shift right | 1 | 1100 1100 | 1010001b |
| | | − 0011 0011 | |
| (*b*) Correction | 1 | 1001 1001 | 1010001b |
| 8. (*a*) Shift right | 1 | 1100 1100 | 11010001 |
| | | − 0011 0011 | |
| | | | ↓ sign bit |
| (*b*) Correction | 1 | 1001 1001 | 11010001 |

Result : $1,1010001 = -1\cdot2^7+2^6+2^4+1$
$= -128+64+16+1$
$= -128+81$
$= -47$

Given decimal number $= 1,53-1.10^8+53$
$= -47.$

In the algorithms discussed for the conversions of signed numbers (in sign-complement form), the arithmetic shifts are to be done on the BCD register. In other words, for shift left operation the sign bit remains unchanged, while most significant bit comes out and gets lost. This lost bit must be same in value as the sign bit, otherwise a conversion overflow occurs and it must be indicated. In shift right operation, the sign bit is retained and is copied in the vacated (most significant) position

## EXERCISES

1. Give a step by step procedure and logic design (using 4-bit binary adders) for a decimal full adder for numbers in excess -3 code.

2. Explain step by step, how the following decimal operations can be carried out for BCD numbers A=298 and B=326.

(*i*) (+A)+(+B) (*v*) (+A)−(+B)
(*ii*) (−A)+(+B) (*vi*) (+A)−(−B)
(*iii*) (−A)+(−B) (*vii*) (+A)+(−B)
(*iv*) (−A)−(−B) (*viii*) (−A)−(+B)

3. Two n-digit decimal numbers are to be added using a single decimal full adder, give the complete logic design to carry out this.

4. In the radix -6 number system, the digits are represented by the 4-2-1 weighted binary code, *i.e.*, digits 0, 1, 2, 3, 4 and 5 are represented by 3-bit binary number 000, 001, 010, 011, and 100, and 101 respectively.

(*a*) Design a logic which finds 6's and 5's of complement of a digit.

(*b*) Design a radix 6 full adder.

5. Digits of numbers in duodecimal (r=12) system are coded by the 8-4-2-1 weighted binary code.

(*a*) Design the 12's complementer for a digit.

(*b*) Design a duodecimal full adder.

6. Repeat (5) for axcess 2 and excess 4 codes.

7. Design a decimal full subtracter for BCD numbers.

8. On the lines of binary to BCD conversion algorithm and viceversa give an algorithm to convert unsigned binary numbers in the weighted 4-2-1 coded radix 6 numbers.

9. Using (8), illustrate the conversion of the unsigned binary number 1011011 into the 4-2-1 coded radix 6 number.

10. Repeat (8) for numbers in sign-complement *i.e.*, give an algorithm to convert binary numbers in sign-complement form into 4-2-1 coded radix 6 numbers in the sign -6's complement form.

11. Repeat (8), (9), (10) to convert 4-2-1 coded radix 6 numbers into binary numbers.

12. Repeat (11) for 8-4-2-1 coded duodecimal numbers.

13. Illustrate the conversion of $235\ A)_{18}$ into binary.

14. Design the combinational and sequential logic circuits for the algorithms developed (8), (10), (11) and (12).

□

# 7

# BINARY MULTIPLICATION AND DIVISION

## 7.1 INTRODUCTION

In this chapter, we shall study the algorithms to carry out multiplication and division operations on binary integers. These algorithms could be implemented by sequential logic circuits or software programs. The algorithms are based on repetitive use of ADD/SUB and shift primitives. In the Section 7.2 various algorithms are given for binary multiplication which require different execution times depending upon the number of bits analysed. Also extensive coverage is given to multiplication of signed numbers. Similarly the division algorithms are given for signed and unsigned numbers in Section 7.3.

## 7.2. BINARY MULTIPLICATION

Multiplication process is based on the fundamental fact that a product X.Y of two

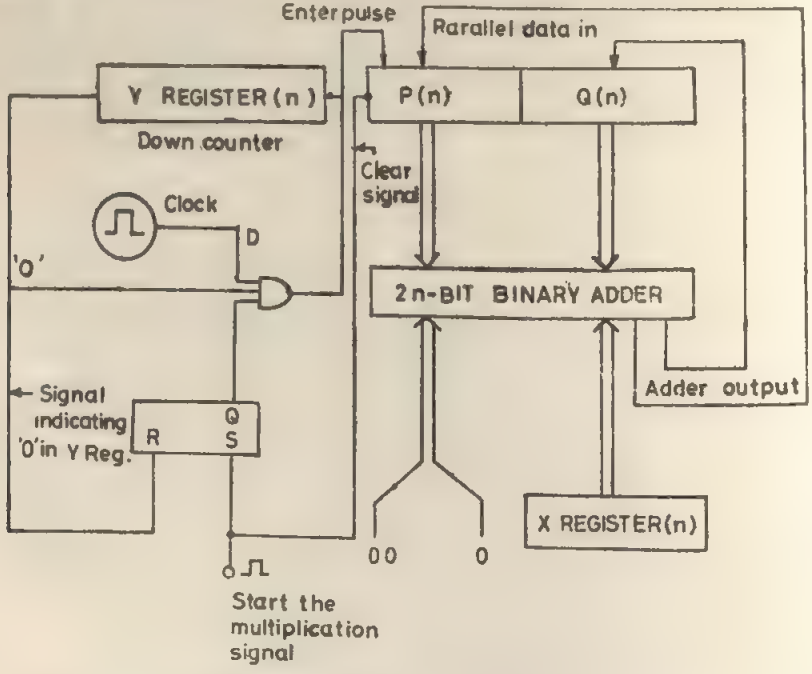



FIG.7-1 SIMPLE MULTIPLICATION LOGIC SCHEMATIC.

integers is nothing but a number P, obtained by adding the number X to itself Y times. The number X is called a multiplicand, while the number Y is called a multiplier. This simple process of obtaining a product P is implemented by the logic circuit schematic of Fig. 7.1. The P register initially contains all zeroes. The 2n-bit adder adds X and P registers and the result is put back in P register. Y register is a down counter to which a down counting pulse is given for every addition of X to P. When the contents of Y register becomes 0, the clock is stopped by resetting the F/F. The P register contains the product.

In this multiplication procedure, the number of steps (*i.e*, time required for the multiplication) is proportional to the value of Y. Moreover, we require a 2n-bit adder. Also, three registers, one with 2n bits and other two with n-bit lengths are required. Due to these reasons, this method is seldom used in practice. In the following sections multiplication algorithms requiring n/k (k=1, 2) steps are discussed.

### 7.2.1. Multiplication By One Bit Analysis Of Multiplier

Let X and Y be two unsigned integers, where

$Y = \sum_{i=0}^{n-1} y_i 2^i$, then, the product

$$X.\ Y = X \sum_{i=0}^{n-1} y_i 2^i$$

$$= Xy_0 + 2Xy_1 + 2^2Xy_2 + \ldots 2^{n-1}Xy_{n-1} \quad \ldots(7.1)$$

Equation (7.1) indicates how to obtain the product XY by the addition of n terms on the right hand side of equation (7.1). Each term on the right handside indicates an addition of lift shifted X by $i$ positions (for $i^{th}$ step) to the partial product (sum of previous terms). This addition is also equivalent to keeping X fixed and shifting partial product by the same amount to the right. The multiplication process is summarised in the algorithm 7.1.

**Algorithm 7.1** : PQ is the product register, P and Q are of n bit length. X is an n-bit register.

X reg←X, : Multiplicand is transferred to X register.

P reg←0, : Partial product register is cleared.

Q reg←Y, : Multiplier is transferred to Q register

Repeat n times the following :

*(i)* If the LSB of PQ register is 1 then P←P+X.

*(ii)* Shift PQ right by 1 bit *i.e*., PQ←PQ/2

**Note** : PQ is the register made by joining P and Q registers. This definition of registers is used throughout this chapter.

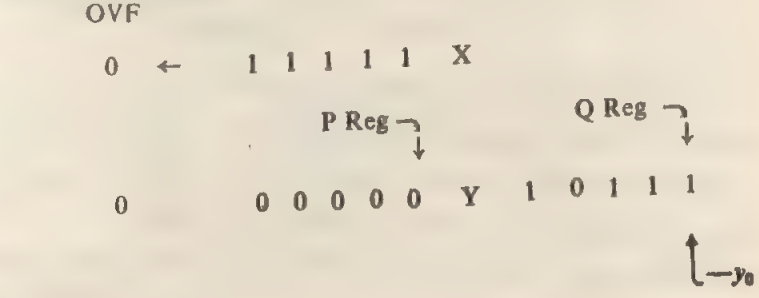
**Example 7.1** : Multiply X=11111 (31) and Y=10111 (23).

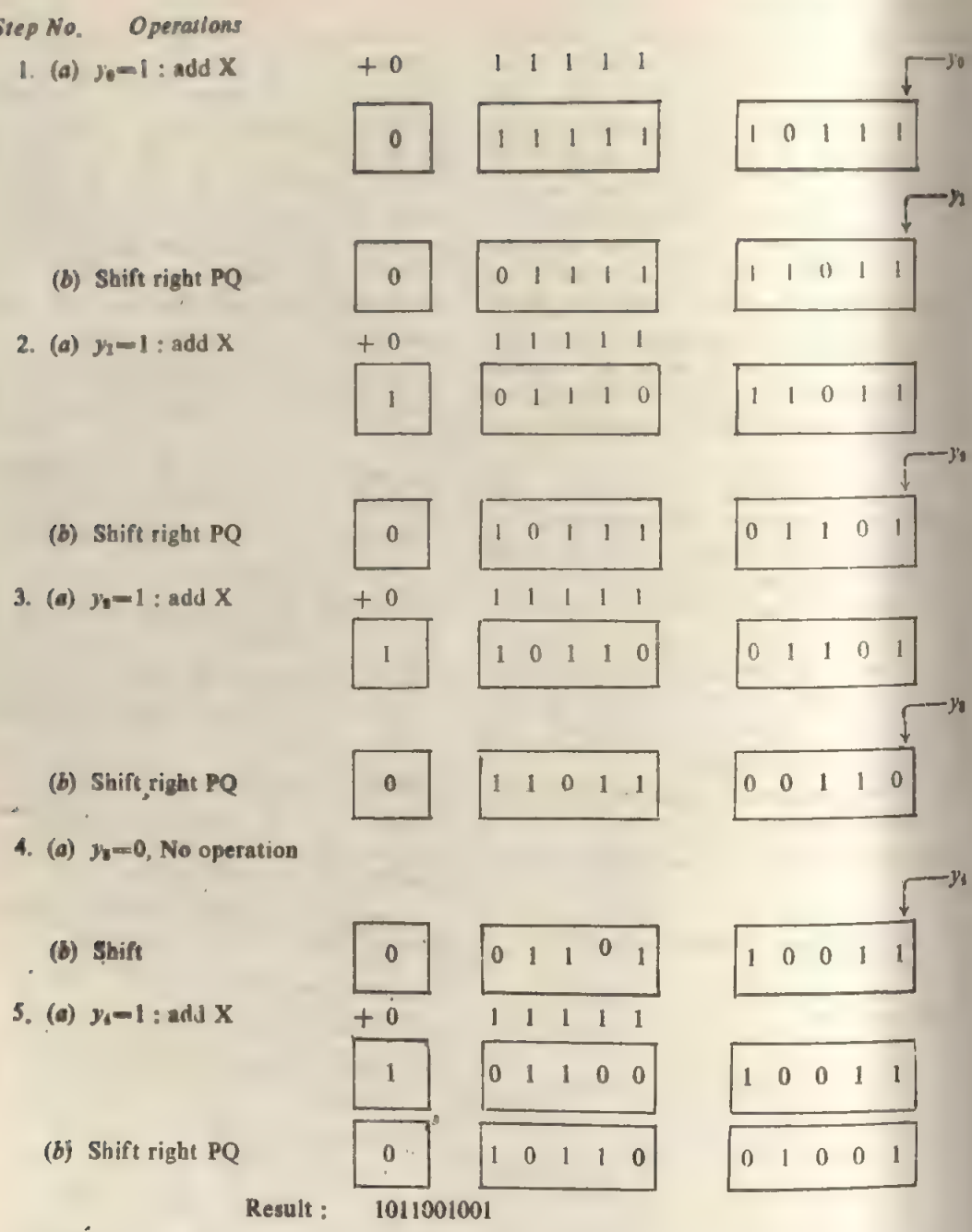
Initial conditions :

*(i)* Transfer X to X register

*(ii)* Clear P register, transfer Y to Q register.

X register—






Result : 1011001001

Algorithm 7.1 requires n steps and makes use of equation (7.1), where X and Y are unsigned numbers. To use this algorithm for multiplying signed numbers in sign-magnitude form, we take magnitude parts, treat them as unsigned numbers and carry out the multiplication. The sign bit of the product is calculated using the relation $p_s = x_s \oplus y_s$, where $p_s$, $x_s$ and $y_s$ are sign bits of product P, the number X and the number Y respectively. The bit $p_s$ is attached to the product to give the signed product.

Multiplication of signed numbers in sign-2's complement form could also be carried

out using Equation (7.1) as discussed below.

The numbers X and Y which are in sign -2's complement form can be expressed by a polynomial in 2 with sign bit having a negative weight (see Chapter 2, Section 2 3).

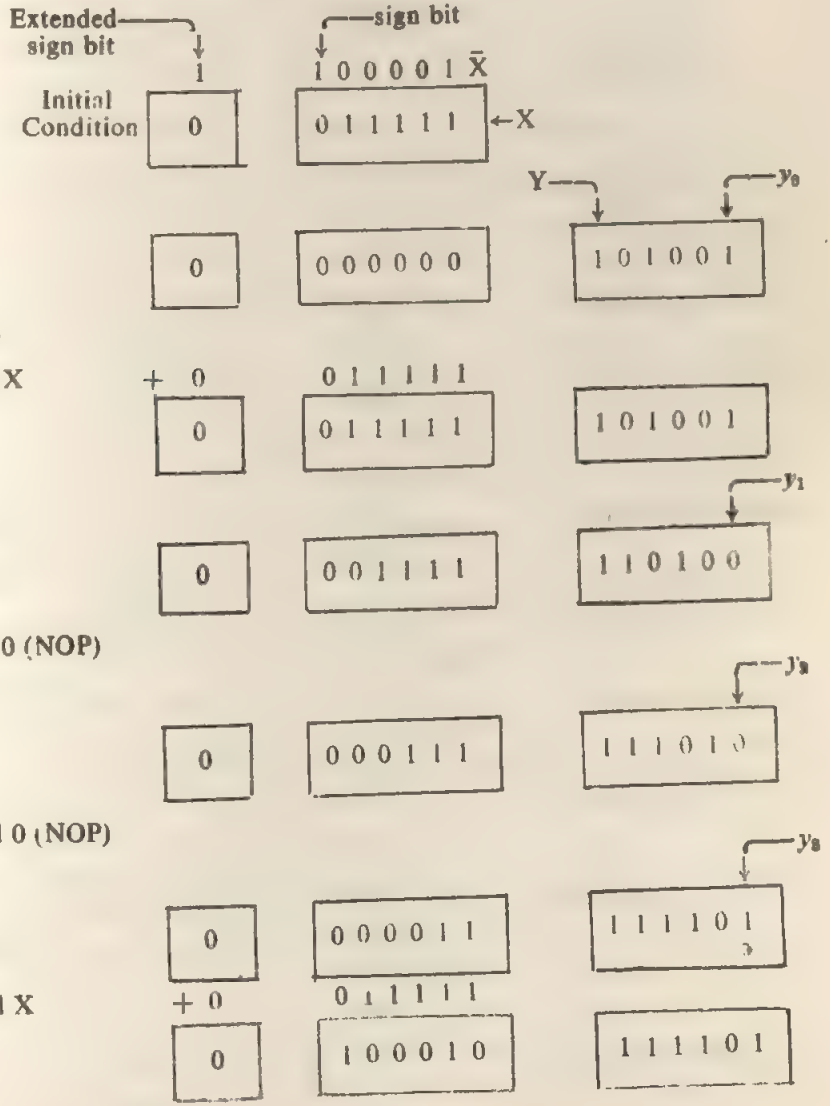
$$X = -x_n 2^n + \sum_{i=0}^{n-1} x_i 2^i$$

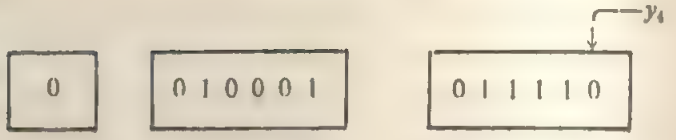
$$Y = -y_n 2^n + \sum_{i=0}^{n-1} y_i 2^i$$

The product $XY = X(y_0 + y_1 2 + \ldots, y_{n-1} 2^{n-1} - y_n 2^n) \ldots (7.2)$

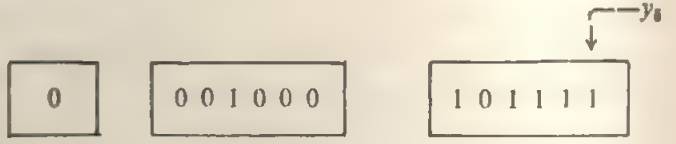
The product expressed here is the result of multiplying two number polynomials of X and Y (with -ve weight to sign positions). This thus will give the required product in sign -2's complement form. Equation (7.2) tells us that the last step will be subtraction of X (instead of addition as indicated in algorithm 7.1). Moreover, the shifting should reflect the division by 2 of the partial product, which is in the sing—2's complement form. Hence, arithmetic shifts have to be employed.

**Example 7.2.** Multiply +31 by −23

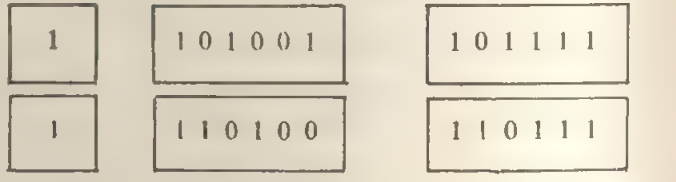





5. (a) $y_4$=0 : add 0

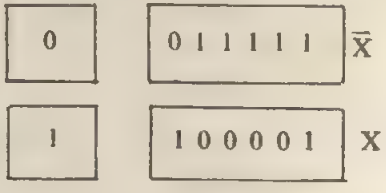


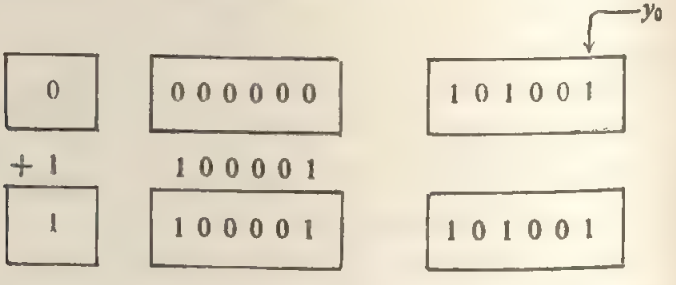

6. (a) $y_5$=1 : (sign bit) add $\bar{X}+1$ 1 0 0 0 0 1

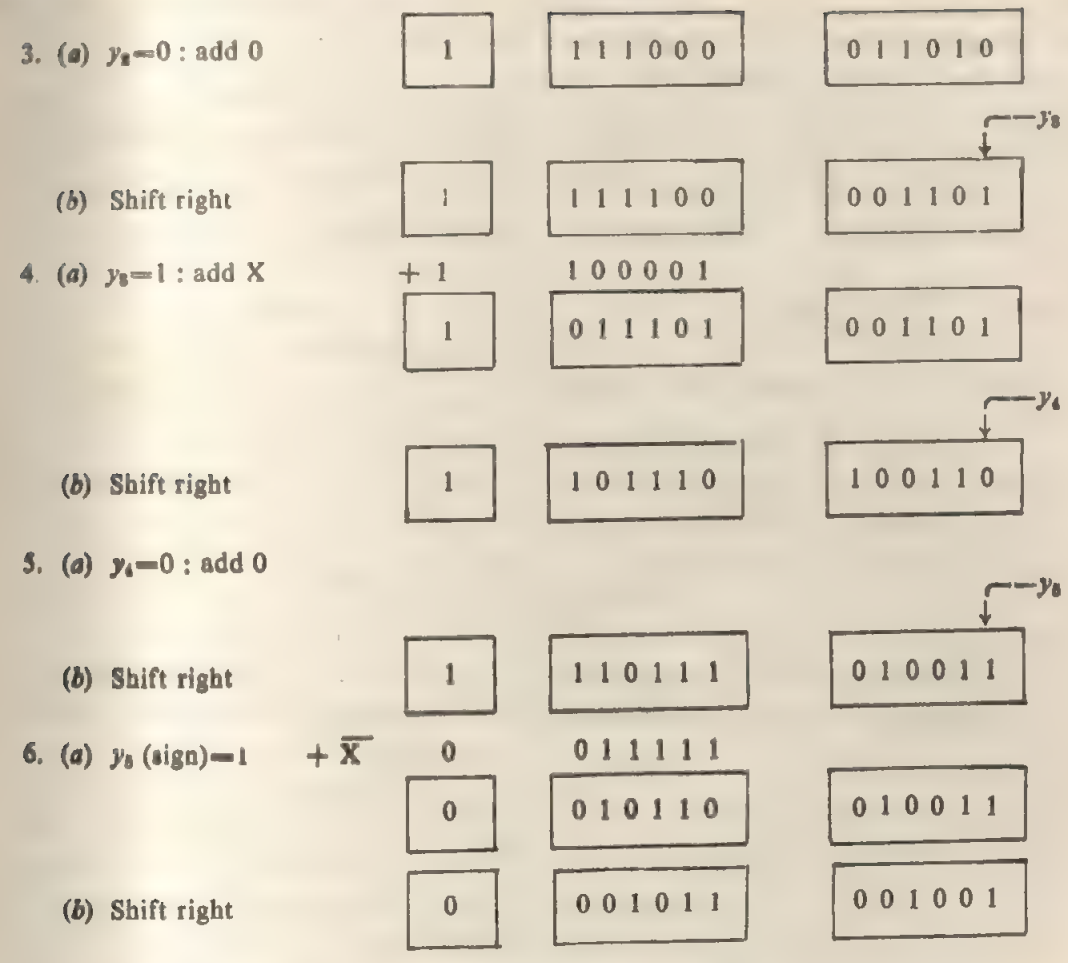



Result : 1,10100110111 (−713).

**Example 7.3** : Multiply X=1.00001 (−31) and Y=1,01001 (23)






Result : 0,01011001001 (+713).

The multiplication procedure discussed in the above examples is summarised in Algorithm 7.2.

Algorithm 7.2 : Multiplication of numbers in sign-2's complement form.

| *Steps* | *Comments* |
|---|---|
| | X, Y, are two n—bit numbers which are to be multiplied. P, Q, X are registers. |
| 1. P←0, Q←Y X←Multiplicand | : Initialise the registers. |
| 2. Repeat n—1 times | |
| 3. If PQ (0)=1 then P←P+X | |
| 4. PQ←PQ/2 | : Arithmetic shift |
| 5. End of repeat | |
| 6. If PQ (0)=1 then P←P—X | : Sign bit analysis |
| 7. PQ←PQ/2 | : Shift PQ right (arithmetic shift) by 1. |
| 8. Stop | |

**Note** : PQ (0) stands for bit 0 of PQ register, *i.e.*, the LSB.

**Booth's Algorithm**

The multiplication methods for sign-complement numbers (equivalent to algorithm 7.2) due to Von Newmann and Goldstine

(1946)* required correction steps to the product obtained in the process of magnitude multiplication. Booth's algorithm does away with these correction steps, morever, it has a uniform structure of steps. We shall analyse this method and illustrate it by an example. However Algorithm 7.2 given here (as a generalisation by Author) is equally good.

Let $Y = -y_n 2^n + \sum_{i=0}^{n-1} y_i 2^i$

Noting that $2^{k-1} = 2^k - 2^{k-1}$, we can write Y as

$$Y = -y_n 2^n + y_{n-1}(2^n - 2^{n-1}) + y_{n-2}(2^{n-1} - 2^{n-2}) + \ldots y_2(2^3 - 2^2) + y_1(2^2 - 2^1) + y_0(2^1 - 2^0)$$

$$i.e., \quad Y = 2^n(y_{n-1} - y_n) + 2^{n-1}(y_{n-2} - y_{n-1}) + \ldots + \ldots 2_2(y_1 - y_2) + 2^1(y_0 - y_1) + 2^0(0 - y_0) \qquad \ldots(7.3)$$

By looking at Equation (7.3) we can say that instead of analysing bits of Y we could as well analyse the two neighbouring bits of Y and accordingly add/subtract X or add 0 to the partial product Table 7 1 gives the list of operations for the $i^{th}$ step depending upon $y_i$ and $y_{i-1}$ in the multiplication process and the algorithm is illustrated by the following example. Note that, to start with we have a $0 - y_n$ step.

TABLE 7.1

| $y_{i-1}$ | $y_i$ | *Operation* | |
|---|---|---|---|
| 0 | 0 | add | 0 |
| 0 | 1 | sub | X |
| 1 | 0 | add | X |
| 1 | 1 | add | 0 |

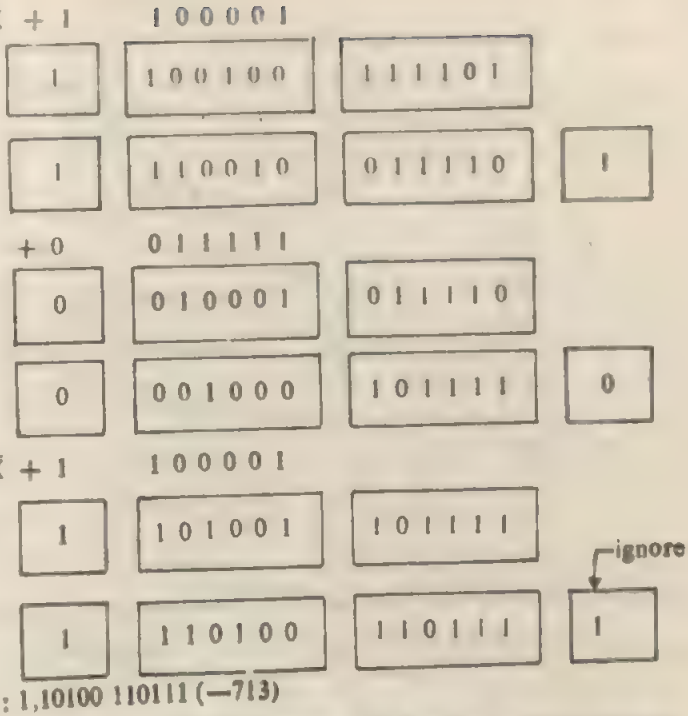
**Example 7.4 :** Illustrate Booth's algorithm by multiplying X = +31 and Y = −23.

1 1 0 0 0 0 1 $\bar{X}$

Initial condition: 0 | 0 1 1 1 1 1 | X — FE stores the shifted bit

| *Steps No.* | *Operations* | P | Q | FF |
|---|---|---|---|---|
| | | 0 | 0 0 0 0 0 0 | 1 0 1 0 0 1 | 0 |
| 1. | (*a*) FF−PQ(0) = −1 : add $\bar{X}$ + 1 | | 1 0 0 0 0 1 | | |
| | | 1 | 1 0 0 0 0 1 | 1 0 1 0 0 1 | |
| | (*b*) Shift right by 1 | 1 | 1 1 0 0 0 0 | 1 1 0 1 0 0 | 1 |
| 2. | (*a*) FF−PQ(0) = 1 : add X | + 0 | 0 1 1 1 1 1 | | |
| | | 0 | 0 0 1 1 1 1 | 1 1 0 1 0 0 | |
| | (*b*) Shift right by 1 | 0 | 0 0 0 1 1 1 | 1 1 1 0 1 0 | 0 |
| 3. | (*a*) FF−PQ(0) = 0 : add 0 | | | | |
| | (*b*) Shift right by 1. | 0 | 0 0 0 0 1 1 | 1 1 1 1 0 1 | 0 |

*See Collected Work of J. Von Newmann Vol, 2.




Result : 1,10100 110111 (—713)

**Note :** Shifts are arithmetic since sign-complement numbers are involved.

The algorithms discussed so far require n steps to multiply a multiplicand with an n-bit multiplier. This is so, because we used only the effect of one bit of Y at a time in the multiplication process. On the contrary if we analyse more bits of Y at a time, corresponding reductions in the number of steps could be achieved.

### 7.2.2 Multiplication by Analysing Multiple Bits Of Multiplier

By analysing more than one bit of a multiplier at a time, the number of steps required to carry out the multiplication can be reduced proportionately. In this section we discuss an algorithm requiring n/2 steps for n-bit multipliers. This multiplication procedure uses 2 bits of multiplier in every step. This, thus, can be thought of as multiplying two numbers in quoternary system (radix 4). At every step we check 2 bits of multiplier, *i.e.*, a digit in quoternary system. Let this digit be d. Then we shall have to add the quantity X. d to the partial product for every step. Since d takes values 00, 01, 10 or 11, this means we have to add 0X, 1X, 2X or 3X to the partial product. Also, shifting is by 2 bits (one digit in quoternary) in every step. Addition of 2X and 3X require availability of 2X and 3X in the arithmetic unit. To avoid the use of more registers for this, we discuss an alternative.

The addition of 2X is equivalent to the addition of left shifted X by one position (usually adders will have provision to select either X or left shifted X as an input). The addition of 3X is achieved by splitting 3X into 4X—X. Thus, if in any step, 3X is to be added we simply subtract X from the partial product and the addition of 4X is done in the next step, which is equivalent to the addition of X itself (this is so because, partial product in the next step is shifted right by 2 places). Table 7.2 lists the operations to be carried out in a step of this multiplication procedure. Third column represents a flag bit which is set to 1 if 4X was required to be added, otherwise it is cleared.

TABLE 7.2

| $y_i$ | $y_{i-1}$ | *4X Flag (F) (old)* | *Effective 2-bit number* $2y_i+y_{i-1}+F$ | | *Operation* | *4 X Flag (new)* |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0+0+0=0 | 00 | add 0 X | 0 |
| 0 | 0 | 1 | 0+0+1=1 | 01 | add 1 X | 0 |
| 0 | 1 | 0 | 0+1+0=1 | 01 | add 1 X | 0 |
| 0 | 1 | 1 | 0+1+1=2 | 10 | add 2 X | 0 |
| 1 | 0 | 0 | 2+0+0=2 | 10 | add 2 X | 0 |
| 1 | 0 | 1 | 2+0+1=3 | 11 | add — X | 1 |
| 1 | 1 | 0 | 2+1+0=3 | 11 | add — X | 1 |
| 1 | 1 | 1 | 2+1+1=4 | (1) 00 | add 0 X | 1 |

**Example 7.5 :** Multiply X=11111 (31) and Y=10111 (23)

| | | | |
|---|---|---|---|
| 1 | 1 0 0 0 0 1 | X | |
| 0 | 0 1 1 1 1 1 | X | |
| 0 | 0 0 0 0 0 0 | | 0 1 0 1 1 1 |

*Steps No.* *Operations*

1. (*a*) add —X (11=100—1) +1 1 0 0 0 0 1
   Flag=1(F=1)

| | | |
|---|---|---|
| 1 | 1 0 0 0 0 1 | 0 1 0 1 1 1 |

(*b*) Shift by 2

| | | |
|---|---|---|
| 1 | 1 1 1 0 0 0 | 0 1 0 1 0 1 |

2. (*a*) add 2X (01+F=01+1 =10) +0 111110
   and F=0

| | | |
|---|---|---|
| 0 | 1 1 0 1 1 0 | 0 1 0 1 0 1 |

(*b*) Shift by 2

| | | |
|---|---|---|
| 0 | 0 0 1 1 0 1 | 1 0 0 1 0 1 |

3. (*a*) add X (01+F=01+0 =01) and F=0 | +0 | 0 1 1 1 1 1 | 1 0 0 1 0 1

(*b*) Shift by 2 | 0 | 0 0 1 0 1 1 | 0 0 1 0 0 1

Result : 001011001001 (713)

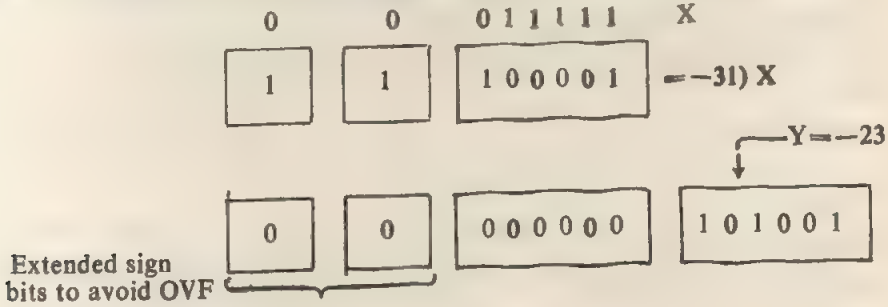
**Note** : In step 1 (*b*) due to borrow in subtraction 1's are copied into vacated position. In general, to avoid the confusion, use two extra bits and treat the left most bit as sign bit and carry out arithmetic shifts.

Multiplication of signed numbers in sign -2's complement form could be carried out using the procedure discussed with minor modification. This is possible due to the fact that the sign bit is to be assumed to have a negative weight and additions/subtractions are on signed number. Since sign bit represents negative weight, the operation in the last step is to be modified accordingly, as shown in table 7.3. Thus, in the last step, we use operations as per Table 7.3.

TABLE 7.3

| $y_s$ | $y_{n-1}$ | *4X Flag (F) (old)* | *Effective 2-bit Number* $2y_s+y_{n-1}+F$ | *Operation* | *4 X Flag (new)* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0+0+0=0 | add 0 X | 0 |
| 0 | 0 | 1 | 0+0+1=1 | add 1 X | 0 |
| 0 | 1 | 0 | 0+1+0=1 | add 1 X | 0 |
| 0 | 1 | 1 | 0+1+1=2 | add 2 X | 0 |
| 1 | 0 | 0 | −2+0+0=−2 | add −2 X | 0 |
| 1 | 0 | 1 | −2+0+1=−1 | add −1 X | 0 |
| 1 | 1 | 0 | −2+1+0=−1 | add −1 X | 0 |
| 1 | 1 | 1 | −2+1+1=0 | add 0 X | 0 |

**Example 7.6** : Multiply −31 with −23 using sign −2's complement representation



| *Steps No.* | *Operations* | | | | |
|---|---|---|---|---|---|
| 1. (*a*) | add X | 1 | 1 | 1 0 0 0 0 1 | 1 0 1 0 0 1 |
| (*b*) | Shift by 2 | 1 | 1 | 1 1 1 0 0 0 | 0 1 1 0 1 0 |
| 2. (*a*) | add 2 X | + 1 | 1 | 0 0 0 0 1 0 | |
| | | 1 | 0 | 1 1 1 0 1 0 | 0 1 1 0 1 0 |
| (*b*) | Shift by 2 | 1 | 1 | 1 0 1 1 1 0 | 1 0 0 1 1 0 |
| 3. (*a*) | add −2 X (last step) | 0 | 0 | 1 1 1 1 1 0 | |
| | | 0 | 0 | 1 0 1 1 0 0 | 1 0 0 1 1 0 |
| (*b*) | Shift by 2 | 0 | 0 | 0 0 1 0 1 1 | 0 0 1 0 0 1 |

Result : 0,01011001001 (+713)

### 7 2.3 Multiple Precision Multiplication

This section discusses a method to carry out the multiple precision multiplication. We can implement the multiple precision multiplication by hardware using a basic facility providing the multiplication of two n-bit numbers. The method depends on elementary relations. It could be best illustrated through an example.

Assume that we have a block which can carry out the multiplication of two 8-bit binary numbers. The multiplication of two 16-bit numbers could be carried out as follows.

Let $X=2^8.x_1+x_0$ and

$Y=2^8.y_1+y_0$ (see Fig. 7.2)

be two 16-bit numbers. Here $x_1$, $x_0$ are the most and least significant 8-bit bytes of X respectively. Similarly $y_1$ and $y_0$ are two bytes of Y. The product P is given by :

$$P=XY$$
$$=(2^8x_1+x_0).(2^8y_1+y_0)$$
$$=2^{16}x_1y_1+2^8(x_1y_0+y_1x_0)+x_0y_0$$

The above equation gives us the method for multiplying two 16-bit numbers. What we need to do is evaluate the elementary products $x_1y_1$, $x_1y_0$, $x_0y_0$, $x_0y_1$ and add them with the appropriate weights as indicated in the above equation. Since the weights are powers of 2, this involves only shifted addition of the elementary porducts. Fig. 7.2 gives the logic diagram of a multiplier which multiplies two 16-bit numbers. This logic uses 16-bit adders and 8x8 multiplier (which multiplies two 8-bit numbers) chips currently available.

Using the 16×16 multiplier, or Fig. 7.2, we can develop the multiplier for multiplying two 32-bit numbers.

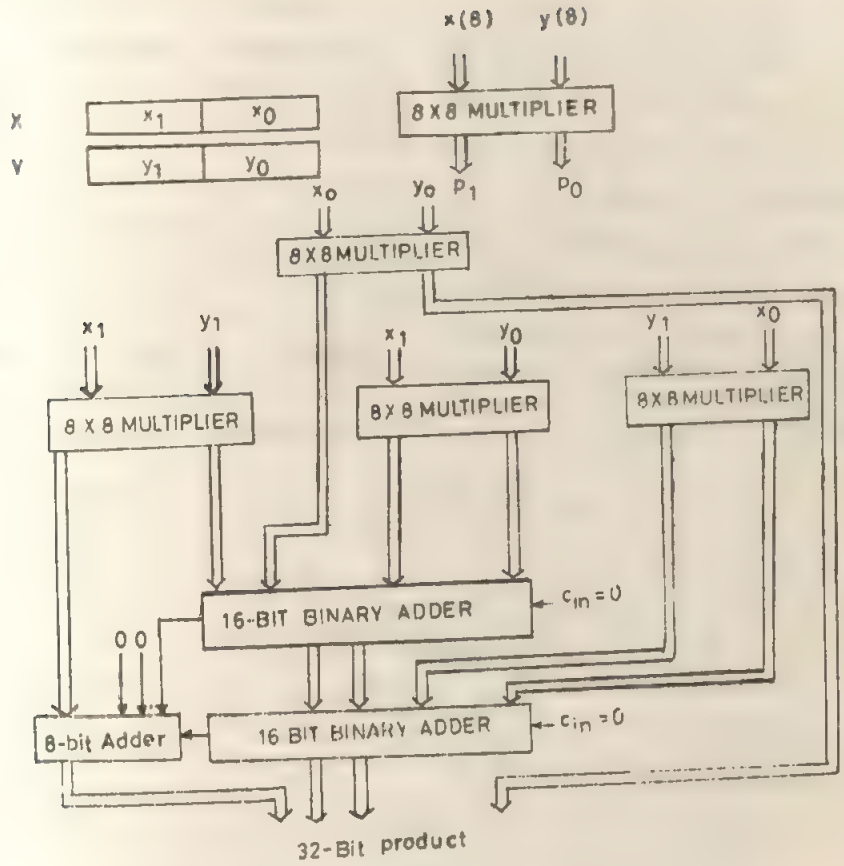



FIG.7·2 16 X 16 MULTIPLIER

## 7.3. BINARY DIVISION

Before proceeding to the division of binary integers, let us see how we carry out decimal division using the arithemetic learned in school days This is discussed below, to arrive at some important conclusions regarding the division process in machines. We illustrate these through the following division.

```
35)4035(1   Quotient digits
 -35
 ---
  053  1
  -35
 ----
  185  5   Lsd
 -175
 ----
  010 ← Remainder
```

In this division, we obtained the quotient Q= 115 and remainder R=10.

In the example discussed the operand 35 is called a divisor while the operand 4035 is called dividend. The quotient and remainder are related as follows :

$X=QY+R$, where X=dividend
Y=divisor
Q=quotient
R=remainder
and $|R|<|Y|$.

In the above example, we divided the 4-digit dividend by the 2-digit divisor. In machines generally we divide 2n-digit dividend by n-digit divisor. and the lengths of the registers holding quotient and remainder are fixed at n. The decimal division in the above

example gave 3 digit quotient and if the quotient register has only 2-digits, the quotient 115 cannot be accommodated and hence an overflow condition should be indicated. The overflow condition occured in this example, because the number 40 formed by 2 significant digits of X is larger than the divisor 35. In general if we are dividing a 2n-digit dividend by n-digit divisor, the number formed by the most significant n digits of a dividend, should be smaller than the divisor and if this is not so an overflow indication has to be given. In such cases we may not proceed further with the division since the result will not be of the interest.

### 7.3.1. Binary Division Using Comparison Method

This method of binary division is exactly same as the one discussed earlier for the example of decimal division. In the binary division we have an added advantage of having a quotient bit either 0 or 1 only and hence either substraction or no subtraction of the divisor is required at every step. Figure 7.3 gives the flowchart for the algorithm and Example 7.7 illustrates it.

**Example 7.7 :** Divide X—01010101 by Y—0110.

The comparison division works as follows

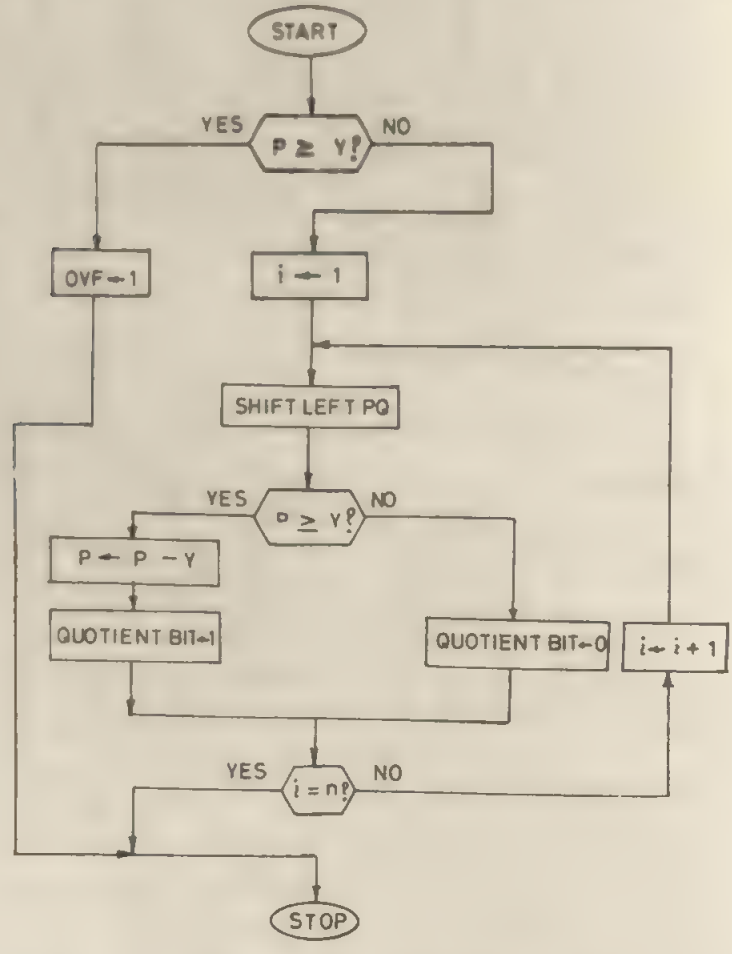
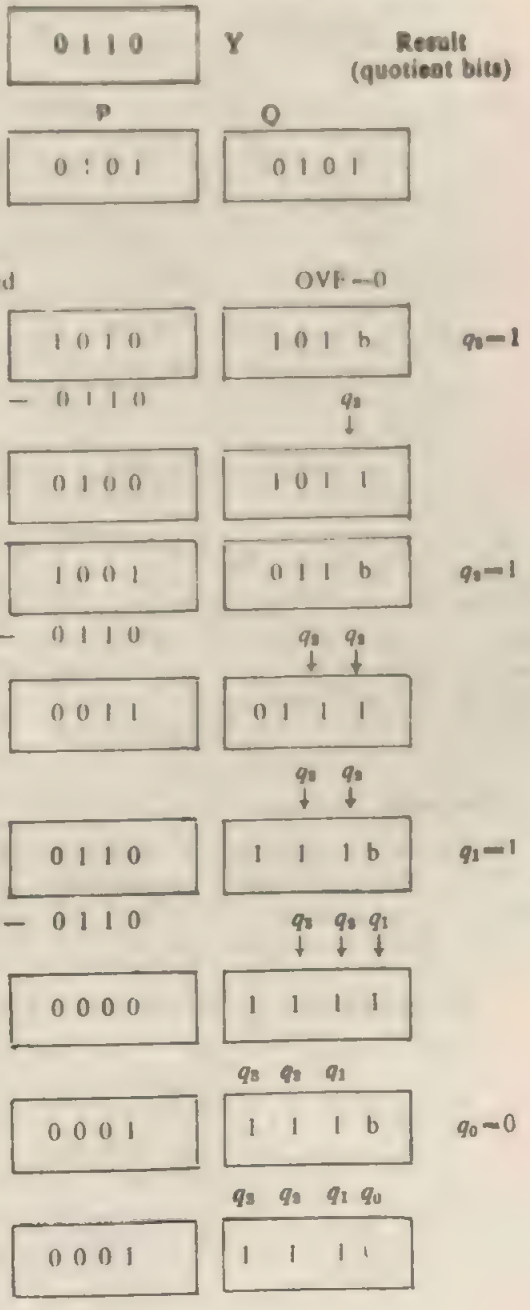



FIG. 7·3 FLOW-CHART OF COMPARISON DIVISION METHOD

We have one register of length 2n bits (PQ register) and a n-bit Y register. At every step, the partial remainder is shifted left by 1 bit and the quotient bit is entered serially in PQ from LSB. The contents of P are denoted by $r_i$ at the end of the $i$th step.




Result : $r_4$=0001 (Remainder)
Q=1110 (Quotient)
OVF=0

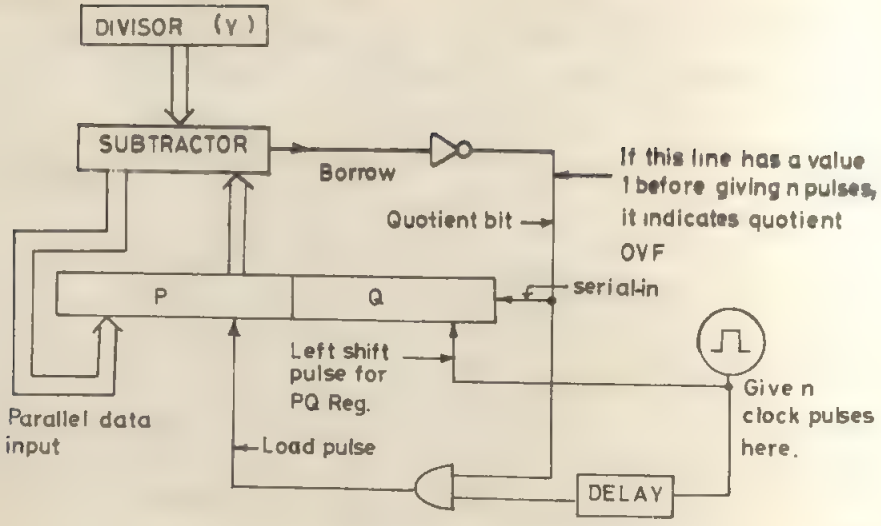


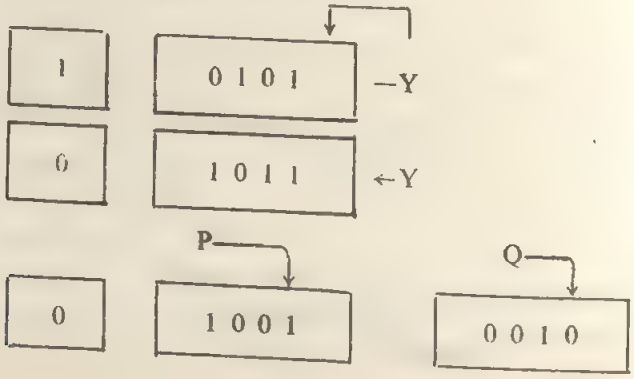

FIG. 7-4 LOGIC CIRCUIT SCHEMATIC FOR COMPARISON DIVISION

The logic circuit using the comparison algorithm for division is shown in the Fig. 7.4. The subtractor is used to subtract the divisor from P register and the result of subtraction is placed back in the P register only if the result of subtraction is positive which is indicated by the absence of the borrow signal in the subtracter.

### 7.3.2. Restoring Division

The comparison algorithm discussed in the earlier section requires a comparison circuit for n-bit operands. In early computers this usually used to be done by subtraction in arithmetic unit. The result of subtraction usually used to be placed in the P register which is normally an accumulator (AC). If the result of subtraction is negative then (difference P—Y is negative) the quotient bit is set to zero and P register has to be restored back by adding Y (since subtraction of Y is required only when q=1). Thus the algorithm steps become non-uniform due to the presence/or absence of restoration. The flow chart for this algorithm (used in second generation computers and called as restoring division) is given in Fig. 7.5 and example (7.8) illustrates it.

**Example 7 8** : Divide X=10010010 by Y=1011.



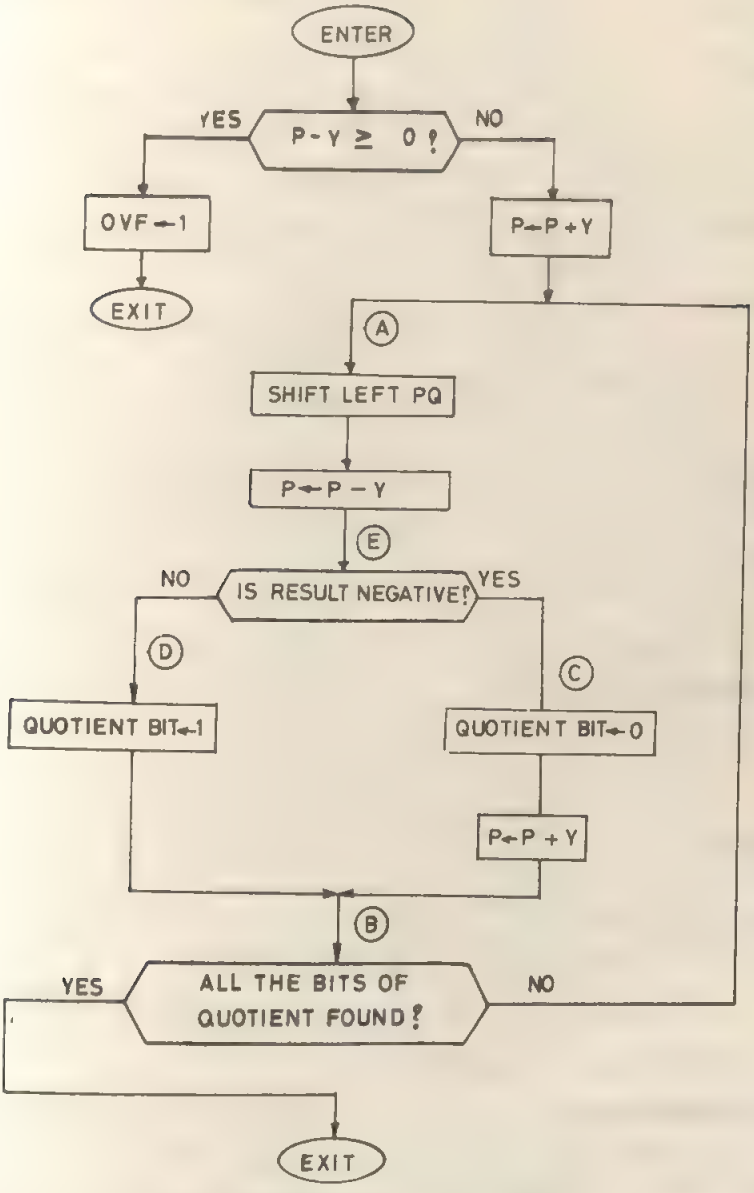
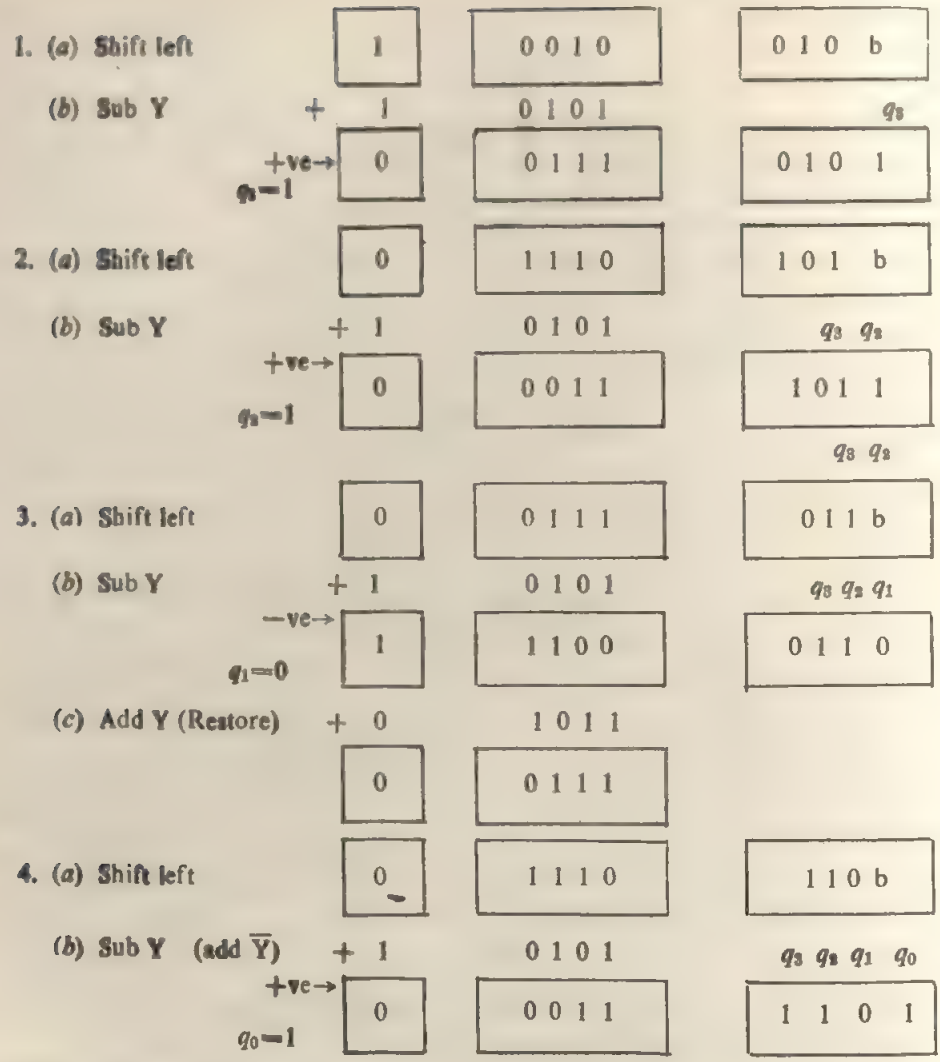



FIG. 7.5 RESTORING DIVISION ALGORITHM

| *Step No.* | *Operations* | | | | |
|---|---|---|---|---|---|
| Q. (*a*) | SUB Y | 1 | 0 1 0 1 | | |
| | Result −ve hence OVF=0 Proceed, | 1 | 1 1 1 0 | 0 0 1 0 | OVF=0 |
| Q. (*b*) | Restore $r_0$ : add Y | + 0 | 1 0 1 1 | | |
| | | 0 | 1 0 0 1 | 0 0 1 0 | |




Result : Q=+1101, R=0011.

### 7.3.3. Non-Restoring Division

The restoring division requires addition of divisor if the result of subtraction were negative. This compels the designer to have three pulses in time sequence for each step *i.e.*, (i) Shift (ii) Subtraction (iii) Restoring. This, thus, not only requires more hardware but also needs more time. To improve on this drawback, a non-restoring division algorithm is proposed which does not require any restore step, but uses either addition or subtraction operations in each step as discussed below. Again this algorithm may not be used in the present day systems because of the availability of number of accumulators.

To analyse how we can proceed without restoring the partial remainder, let us look closely at the flow chart of Fig. 7.5.

In Fig. 7 5, control goes from point A to B and back. Consider the cycles of activities as below.

(a) Flow Cycle AECBA : Let $r_{i+1}$ and $r_i$ be the remainders at the points B and A respectively, Then,

$$r_{i+1} = \underset{\text{Shift}}{2r_i} - \underset{\text{Sub Y}}{Y} + \underset{\text{Restore}}{Y} = 2r_i \qquad \ldots(7.4)$$

Suppose that we do not want to add Y as indicated in the path CB, then the remainder say $r_{i+1}'$ at B will be :

$$r_{i+1}' = 2r_i - Y$$

*i.e.*, $r_{i+1}' = r_{i+1} - Y$ (from Equ. 7.4)

or $r_{i+1} = r_{i+1}' + Y$ ...(7.5)

$r_{i+1}'$ is the unrestored remainder at B and $2r_i$ is the restored remainder (which should be used for the next step). Proceeding to the next step (*i.e.*, going back to A) and working with $r_{i+1}'$ in place of $2r_i$, we have :

| | | On actual remainder | On unrestored remainder |
|---|---|---|---|
| (i) | Shift | $2r_{i+1}$ | $2r_{i+1}'$ |
| (ii) | Sub Y | $2r_{i+1} - Y$ | $2r_{i+1}' - Y$ |

...(7.6)

But $r_{i+1} = r_{i+1}' + Y$

Hence $2r_{i+1} - Y = 2r_{i+1}' + 2Y - Y$ (from Equ 7.6)

$$= 2r_{i+1}' + Y$$

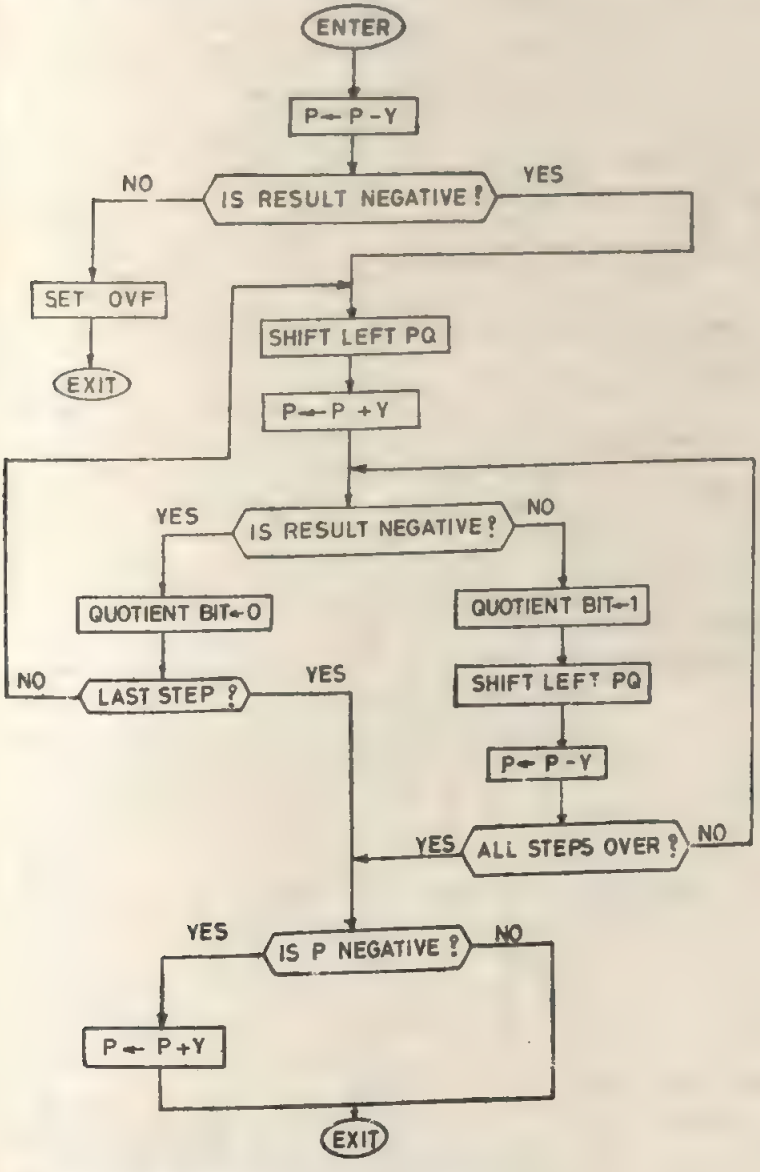



FIG.7.6 NON-RESTORING DIVISION ALGORITHM

The equation (7.6) tells us a very interesting fact. It indicates the possibility of obtaining the same result while working with an unrestored remainder, provided subtract Y operation is replaced by add Y operation in the next step.

**(*b*) Flow Cycle AEDBA**

In this case $r_{i+1}$ and $r'_{i+1}$ are same and the usual process of the flow chart of Fig. 7.5 follows.

From the above discussion, a point to be noted is that if the result of the last operation is negative, the quotient bit is zero and the next operative is addition. Otherwise, the quotient bit is 1 and subtraction is the operation to be performed in the next step. Fig. 7.6 gives the flow chart for this method.

**Example 7.9.** Divide X=10010010 by Y=1011.

| | | | | |
|---|---|---|---|---|
| | | 1 | 0101 | $\bar{Y}$ |
| | | 0 | 1011 | Y |
| | | | P → | Q → |
| | | 0 | 1001 | '0010 |
| *Steps No.* | *Operations* | | | |
| 0 | Add $\bar{Y}$ + 1 | | 0101 | $OV_F$=0 Proceed |
| | Result —ve | 1 | 1110 | 0010 |
| | Result —ve, Proceed | | | |
| 1. | (*a*) Shift left | 1 | 1100 | 010 b |
| | (*b*) Last Result —ve, Add Y+0 | | 1011 | $q_0$=1 |
| | Result +ve | 0 | 0111 | 010 1 |
| 2. | (*a*) Shift left | 0 | 1110 | 101 b |
| | (*b*) Last Result +ve : Sub Y+1 | | 0101 | $q_0$=1 |
| | Result +ve | 0 | 0011 | 10 1 1 |
| 3. | (*a*) Shift left | 0 | 0111 | 011 $b_0$ |
| | (*b*) Last result +ve : sub Y+1 | | 0101 | $q_1$=0 |
| | Result —ve | 1 | 1100 | 011 0 |

4. (*a*) Shift left

| | | |
|---|---|---|
| 1 | 1000 | 110 b |

(*b*) Last result −ve : Add Y+0 1011

| | | | |
|---|---|---|---|
| Result+ve→ 0 | 0011 | 110 1 | $q_0=1$ |

Results : Q=1101, R=0011.

**Note** : Last result' stands for the result of addition/subtraction in the previous step (prior to shift operation of the current steps).

We have discussed various algorithms which operate on given unsigned integers. The division of signed numbers can be carried out using these algorithms if the numbers are in sign-magnitude form by taking only the magnitude parts and operating on them using these algorithms. The signs of quotient and remainder could be attached later using the following relations :

$$r_s=x_s$$
$$q_s=x_s\oplus y_s$$

Where $r_s$, $x_s$, $q_s$ and $y_s$ are the sign bits of remainder, dividend, quotient and divisor respectively.

Von Newmann* and others proposed the non-restoring division algorithm to operate on the numbers in sign-2's complement form. The algorithm and the analysis of the proof is as follows

In this algorithm, divisor is either added or subtracted from the partial remainder depending on the signs of the two. If these two signs agree, the divisor is subtracted and the quotient bit is set to 1 else they are added and the quotient bit is set to 0. In either case, the next partial remainder is obtained by shift left, and the process continues.

Let $r_0$ be the initial remainder (contents of P register) then

$r_1=2r_0+Y$ if $q_n=0$ and

$r_1=2r_0-Y$ if $q_n=1$, combining these two equations, we have

$$r_1=2r_0+(1-2q_n).\ Y \qquad ...(7.7)$$

Where $q_n$ is the first quotient bit in the position $n$ (MSB *i.e.*, sign bit). Similarly

$$r_2=2r_1-(1-q_{n-1})\ Y$$
$$=2(2r_0+(1-2q_n)\ Y)-(1-2q_{n-1})\ Y$$
$$=2^2r_0+2\ (1-2q_n)\ Y+(1-2q_{n-1})\ Y$$
$$r_3=2r_2+(1-2q_{n-2})\ Y$$
$$+2^2\ (1-2q_n)\ Y+2^1\ (1-2q_{n-1})\ Y$$
$$+2^0\ (1-2q_{n-2})\ Y$$

and

$$r_4=2^4r_0+2^3\ (1-2q_n)\ Y+2^2\ (1-2q_{n-1})\ Y$$
$$+2^1\ (1-2q_{n-2})\ Y+2^0\ (1-2q_{n-3})\ Y$$

In general the remainder $r_k$ is given by :

$$r_k=2^kr_0+Y\sum_{i=1}^{k}2^{i-1}\ (1-2q_i)$$

At the last step k=n, we have :

$$r_n=2^nr_0+Y\sum_{i=1}^{n}2^{i-1}\ (1-2q_i) \qquad ...(7.8)$$

The relation between X, Y, Q and R is :

$$X=QY+R, \text{ i.e.,} \qquad ...(7.9)$$
$$R=X-QY \qquad ...(7.10)$$

$2^nr_0$ is the number formed by register P because P is n bits left to LSB. This approximately equals X (error=initial Q register contents, which results in an error of at most 1 in Q).

Thus, $2^nr_0\cong X$ ...(7.11)

Substituting (7.11) in equation (7.8) we get :

$$r_n=X+Y\sum_{i=1}^{n}2^{i-1}\ (1-2q_i)$$
$$=X+Y\sum_{i=1}^{n}2^{i-1}-Y\sum_{i=1}^{n}2^{i-1}.\ 2q_i$$
$$=X+Y\sum_{i=1}^{n}2^{i-1}-Y\sum_{i=1}^{n}2^iq_i \qquad ...(7.12)$$

The term $\sum_{i=1}^{n}2^iq_i$ is the number formed by the quotient bits entered in the Q register. Note that $q_0$ is not found by this

*See collected work of J. Von Newmann Vol. 2.

process. Moreover, comparing equation (7.12) with equation (7.10) the actual quotient is not the one obtained in the Q register. The correction to be made to the quotient is given by the term :

$$-\sum_{i=1}^{n} 2^{i-1}$$

Noting that $-\sum_{i=1}^{n} 2^{i-1} = -2^n+1$, we have,

Therefore,

$$R=r_n=X+(1-2^n)\,Y-Y\sum_{i=1}^{n} 2^i q_i$$

$$=X-Y\left[\sum_{i=1}^{n} q_i 2^i +1-2^n\right]$$

Thus, the actual quotient (terms in bracket) is obtained by adding $(1-2^n)$ to the quotient (pseduoquotient) obtained in Q register. This amounts to the subtraction of a 1 from the MSB and the addition of a 1 to the LSB of the pseduo quotient obtained. Note that the least significant bit of the quotient obtained by this algorithm (after correction) is always 1. This may or may not be the actual case. This arises partly due to the approximation in equating X and $2^n r_0$.

Note that quotient and remainder obtained by this algorithm are in sign—2's complement form.

**Example 7.10** Divide X=1,10010101 by Y=0, 1101.

| *Step No.* | *Operations* | | | | |
|---|---|---|---|---|---|
| | | | 1 | 0011 $\bar{Y}$ | |
| | | | 0 | 1101 Y | |
| | | $r_0$ | 1 | 1001 | 0101 |
| 1. | (*a*) Shift | | 1 | 0010 | 101 b |
| | (*b*) $r_0$—ve, Y +ve add Y+0 | | | 1101 | $q_8$=0 |
| | | $r_1$ | 1 | 1111 | 1010 |
| 2. | (*a*) Shift | | 1 | 1111 | 010 b |
| | (*b*) $r_1$ —ve, Y +ve Add Y+0 | | | 1101 | $q_9$=0 |
| | | $r_2$ | 0 | 1100 | 0100 |
| 3. | (*a*) Shift | | 1 | 1000 | 100b |
| | (*b*) $r_2$ +ve, $\bar{Y}$ +ve Add $\bar{Y}$ | | 1 | 0011 | $q_{10}$=1 |
| | | $r_3$ | 0 | 1011 | 1001 |

4. (*a*) Shift

| | | | |
|---|---|---|---|
| | 1 | 0111 | 001b |

(*b*) $r_3$ +ve and $y$ +ve : add $y$+1 0011

$q_1$=1

| | | | |
|---|---|---|---|
| $r_4$ | 0 | 1010 | 0011 |

Results : R=1010

pseduo quotient=0, 011b

+1, 0001

1, 0111 (−9)

The relation X=QY+R is satisfied as follows :

$-107=-9\times13+10$.

The quotient, obtained in the obove example is still inaccurate, and it is due to the way the equality X=QY+R is satisfied, *e.g.*, (−107=−9×13+10) it is possible to increment/decrement the quotient appropriately by looking at the signs of X, Y, Q and R. The exact logic formuiation for this is left to the readers as an exercise.

### 7.3.4. Multiple Precision Division

In this section, we discuss the multiple precision division. The basic process involved in the multiple precision division of integers is same as that discussed for the single precision integers. Only difference is that, a step for obtaining a bit of the quotient in the single precision division, corresponds to a step for obtaining a quotient word. Moreover, in the earlier case (single precision binary division), the quotient digit $q$ could be 0 or 1, and therefore the subtraction of $q$.Y is equivalent to subtraction of zero or subtraction of Y. In the multiple precision case, $q$ shall be a quotient word and hence we require to multiply $q$ and Y. The quotient word $q$ here is obtained by dividing the most significant two words of X by most significant one word of Y. The quantity $q$.Y is then subtracted from partial remainder (initially most significant part of X). The result of this subtraction can be positive or negative. If the result is negative, the quotient guessed is higher than the actual quotient. The proper quotient word is obtained by successively adding Y to the partial remainder (at the same time the quotient word is decremented by one for each addition) till the remainder is positive. Note that initially the most significant half part of X should be smaller than Y to avoid the quotient overflow. Moreover, whenever the single precision division, (during any step of guessing the quotient word) gives the overflow due to zero in the most significant word of Y, the quotient guess should be made using the next lower order non-zero word of Y.

**Example 7.11** : Illustrate the division of X=40329162 by Y=5169.

| | | | |
|---|---|---|---|
| | | 5169 | Y |
| 0 | 4032 | 9162 | X |

*Steps*

0. 4032 < 5169, NO OVF, hence proceed

1. (*a*) Shift X 4 0329 162*b* $q$=Quotient of 40/5=8
   (*b*) Sub $q$.Y −4 1352 8×5169=41352

   borrow (−1) 9 8977 162*b*

   Add Y +0 5169 $q$=8−1=7

   +ve→0 4146 $q$=7

2. (*a*) Shift X 4 1461 62*bb* $q$=Quotient of 41/5=8
   (*b*) Sub $q$.Y −4 1352 8×5169=41352

   +ve→0 0109 62*bb* $q$=8

3. (*a*) Shift X 0 1096 2*bbb* $q$=Quotient of 01/5=0

   $q$=0

4. (*a*) Shift X 1 0962 *bbbb* $q$=Quotient of 10/5=2
   (*b*) Sub $q$.Y −1 0338 2×5169=10338

   +ve→0 0624 $q$=2

**Results :** Q=7802 R=0624

We illustrated the procedure of multiple precision division through the decimal example. It should be noted that the division process used :

(i) A basic facility of division involving the division of a 2-digit number by a 1-digit number and

(ii) Multiplication of Y with quotient digit guessed (this can be done using multiple precision multiplication).

(iii) Multiple precision subtraction.

(iv) Shift operation.

In general a digit in the above example can be replaced by a word, and the facilities listed above can be appropriately changed to operate on words.

## EXERCISES

1. Illustrate the multiplication of unsigned binary numbers X=01101, Y=10111 using 1-bit analysis of Y at a time.
2. Repeat (1) using 2-bit analysis.
3. Illustrate the multiplication X=1,10011, by Y=1,01101 using 1-bit analysis of Y (X and Y are in sign-2's complement form).
4. Illustrate the comparison division algorithm by dividing X=01001011 by Y=0101.
5. Repeat (4) for non-restoring division.
6. Illustrate the comparison divisions algorithm by dividing X=·01010 by Y=·1011.
7. Repeat (6) for non-restoring division.

8. Repeat (3) using 2-bit analysis of Y.

9. Illustrate the Booth's algorithm by multiplying X=1,11011 with Y=1,01101.

10. Using single digit decimal multiplication facility (assume a block which carries out multiplication of two one digit decimal numbers to produce 2-digit product), give the block diagram design of a decimal multiplier for multiplying 4-digit decimal numbers.

11. Give the flow-chart of the algorithm in (10) so that two n-digit numbers could be multiplied.

12. Illustrate the double precision division by dividing X=1010 1101 0110 1101 by Y=1011 0111 Assume that you have single precision division (division of 8 bit dividend by 4-bit divisor) and double precision multiplication facilities.

13. Similar to what we did for 2-bit analysis of Y in the multiplication algorithm, devise a multiplication algorithm, which can analyze 3 bits of Y at a time.

14. Illustrate the algorithm suggested by you in (13) by multiplying X=011101110 by Y=101101110.

□

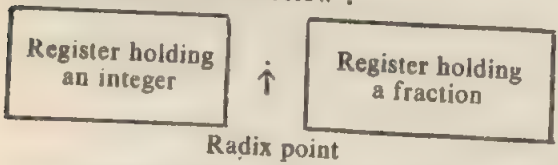
# 8

# FLOATING POINT ARITHMETIC

A floating point number has two parts, viz., a mantissa (usually a fraction) and an exponent (an integer). To carry out an arithmetic operation on floating point numbers, we have to perform arithmetic operations on fractions (mantissa) as well as an integers (exponents). Section 8.1 introduces the arithmetic on fractions and remainder of this chapter is devoted to floating point arithmetic.

## 8.1. Arithmetic Operations On Fractions

Unlike unsigned integers, unsigned fractions have digits only or right hand side of the radix point as shown below :




A signed fraction N shall be denoted as :

$$N = a_s\, a_{n-1} a_{n-2} \ldots a_1 a_0$$

where N is the signed fraction having sign bit $a_s$ and magnitude bits as $a_{n-1}, a_{n-2}, \ldots a_0$. The .(dot) between $a_s$ and $a_{n-1}$ represents the radix point.

### 8.1.1. Addition And Subtraction

The addition and subtraction operations on fractions can be carried out as they are done on integers. The radix point remains unaltered in the result.

**Example 8.1** : Add A=1.101 and B=1.110. (Assume A and B in sign-2's complement form).

| | |
|---|---|
| A=1.101 | (−3/8) |
| B=1.110 | (−2/8) |
| S=1.011 | (−5/8) |
| OVF=0 | |

### 8.1.2. Multiplication

Let $X_f = x_s.x_{n-1}x_{n-2}\ldots x_0$

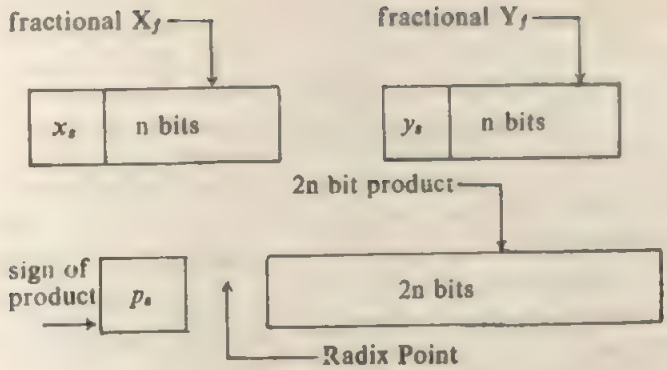
$Y_f = y_s.y_{n-1}\ldots y_0$, be

two signed fractions to be multiplied. Let us form numbers $X = 2^n.X_f$ and $Y = 2^n Y_f$. These numbers are integers with same bit patterns as original $X_f$ and $Y_f$. Since X and Y are integers, they can be multiplied using any of the algorithms discussed in Chapter 7. Let P be their product. The actual fractional product can be obtained as follows :

$$P = X.Y = X_f.Y_f.2^{2n}$$

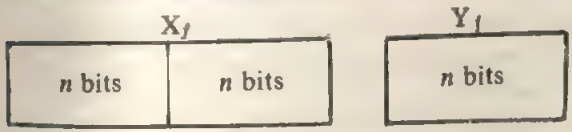
Thus, $X_f.Y_f = \dfrac{P}{2^{2n}}$

Thus radix point of the product is $2_n$ places to the left of least significant bit.




### 8.1.3. Division

Let $X_f$ and $Y_f$ are fractional dividend and divisor of length (excluding the sign bits) 2n and n bits respectively.




Assuming the radix points on the right hand sides, the registers hold numbers $X_f.2^{2n}$ and $Y_f.2^n$. These are integers and can be divided using integer division algorithms. The radix point for the quotient and remainder could be obtained as follows :

$$X_f.2^{2n}=Q.2^n.Y_f+R$$

Hence $$X_f = Q.2^{-n}.Y_f+R.2^{-2n}$$
$$=Q_f.Y_f+R_f$$

The quotient Q obtained in the integer division can be considered as fractional quotient by assuming the radix point at the left side ($Q.2^{-n}$). The remainder is given by $R.2^{-2n}$, where R is the integer remainder obtained in the assumed integer division.

**Example 8.2** : Consider the division of $X_f$=·01010101 by $Y_f$=·0110

The integers taking part in the division are $X_f2^{2n}$=01010101 and $Y_f.2^n$=0110.

The quotient Q and remainder R obtained in the division shall be :

Q=1110 and R=0001.

The fractional quotient and remainder are :

$$Q_f=Q\times 2^{-n}=\cdot 1110$$
$$R_f=0001\times 2^{-8}$$
$$=\cdot 0001\times 2^{-4}$$

### 8.2. Floating Point Numbers

The word Floating Point refers to the radix point whose position is not fixed but floating *i.e.*, may differ with various numbers. For example, 132·10 and 1·821 are two numbers whose decimal points are at different positions. Floating point number representation in machines tries to approximate the real number system of mathematics. Consider a physical situation where the masses of stars and elementary particles are involved in computations. Let the mass of a star be $4\cdot 10^{33}$ gms., while that of electron be $2\cdot 10^{-28}$ gms. All the calculations on such numbers can be carried out using fixed point number system having 34 and 29 digits in its integer and fractional parts respectively. Doing so would allow the representation of numbers with 63 significant digits and the range of $10^{34}$ could be achieved.

On binary computers multiple precision arithmetic (use of more than one computer word for the number representation) could be used to provide the required precision. However, the mass of an electron or a star is not known accurately even to the five significant digits, let alone 63. Although it would be possible to keep all the significant digits in calculations, we will have to throw away the last 50 to 60 digits before printing of the final results.

With the above fixed point representation, there is a considerable wastage not only of computer time but also of the memory. What is needed is a system for representing numbers in which the range of a number system is independent of the significance. One way of separating the range and precision is the scientific notation.

$$N = f.10^{\pm e}$$

Where $f$ = mantissa (usually a fractional number)

$e$ = integer (usually an integer).

In such a representation, the fraction 'f' holds a given number to the required (or available) precision, while available range of 'e' determines the range of the number system. In a general radix r, a floating point number N is represented by $N = f.r^e$. Since we assumed the mantissa to be fractional, implying a fixed radix point to left, the number $f.r^e$ merely means that our number N is f itself with radix point shifted to the left or right by e places depending on whether, e is negative or positive respectively. Since the radix point of N is no longer fixed but is decided by e we call the representation as floating point representation.

**Example 8.3 :** $x = \cdot 2354 \times 10^2$ represents 23·54

$y = \cdot 2354 \times 10^{-3}$ represents ·0002354.

In digital computers the number of digits in 'f' and 'e' have to be decided during its design depending on the possible use of a computer In general purpose computers, mantissae my have 5 to 12 digits, depending on the word length of a computer. Exponent is generally 2 to 3 digits in length.

**Examples Of Floating Point Formats**

Floating point number representations in HP 2100 series and IBM 360/370 series of computers :

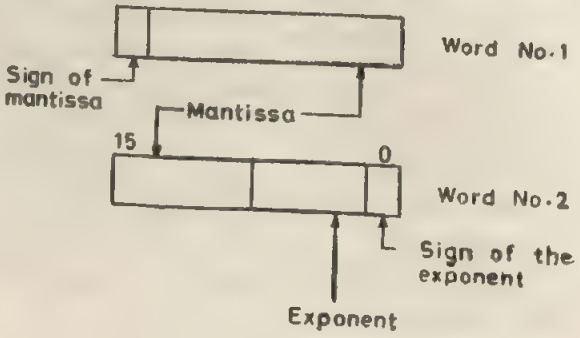
(*a*) **HP 2100 series :** HP 2100 series minicomputers uses two words (word length is 16 bits) for single precision floating point representation. The format is as given below :

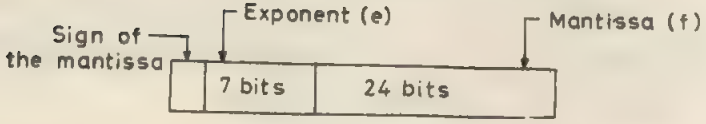
The mantissa and exponents are signed numbers and they are represented in sign —2's complement form. The radix assumed is 2 and the range of numbers is :

$$\pm 2^{+128} \quad \text{to} \quad \pm 2^{-128}$$

(*b*) **IBM 360 series :** Single precision floating point format for IBM 360/370 series of computers (word length is 32 bits) is shown below :

The value of the number represented is given by $\pm f.16^{e-64}$. Note that 'e' does not have a sign bit, the number represented by bits for 'e' actually represents an exponent of



value e—64 (excess 64). Also, the base is not 2 but 16. This enlarges the range of exponent by a factor of 4 over that with the conventional radix 2 system.

## 8.3. FLOATING POINT ARITHMETIC PRINCIPLES

Since floating point numbers have two parts viz., mantissa and exponent, the arithmetic operations and their implementations depend upon the machine representation of fractions and exponents.

Fractions are signed fixed point numbers, whose radix point is on the left hand side of the most significant digit. They can have any of the signed number representation, *i.e.*,

(*i*) Sign—Magnitude

(*ii*) Sign—r's complement

(*iii*) Sign—(r—1)'s complement.

The arithmetic operations on fractional numbers have been briefly discussed in section 8.1. These operations can be implemented in the same manner as is done for integers.

The exponents can be represented in sign—magnitude or sign—complement form. Also biased (excess) form is common for exponent representation. Biased representation of exponent allows representation of both positive and negative exponents by unsigned integers. The bias value is usually chosen such that an entire range of unsigned exponents is equally divided between positive and negative values. For example, if we have a 2-digit exponent in decimal base, then the number of possible states are 00 to 99, *i.e.*, 100. To divide them equally, we have a bias of 50. Similarly biased binary exponents of n-bit length have a bias of $2^{n-1}$.

### 8.3.1. Floating Point Addition/Subtraction

Let $x_1 = f_1 \cdot r^{e_1}$ and

$x_2 = f_2 \cdot r^{e_2}$

be two floating point numbers in the radix $r$. The addition or subtraction is carried out as follows :

*Case I* : If $e_1$ and $e_2$ are equal, then the sum/difference of these numbers ($S = f_s \cdot r^e$) is obtained as follows :

(i) $e \leftarrow e_1$

(ii) $f_s \leftarrow f_1 \pm f_2$

(iii) If no overflow (OVF) in addition/subtraction of $f_1$ and $f_2$ then stop, else shift right $f_s$ with carry and increase the exponent and stop.

**Example 8.4** : Add $X_1 = \cdot 2310 \times 10^3$ and $X_2 = \cdot 8921$

$$\begin{array}{r} 0{\cdot}2310 \\ +0{\cdot}8921 \\ \hline 1{\cdot}1231 \end{array}$$

carry = 1, OVF = 1

after shifting carry in to $f_s$ we have :

$f_s = \cdot 11231$ and

$e = e_1 + 1 = 4$ and the sum is given by :

$S = \cdot 11231 \times 10^4$

**Case II : $e_1 \neq e_2$.**

If $e_1$ and $e_2$ differ, it is not possible to add fractions unless the weightages (exponents) of the fractional digits, taking part in addition/subtraction, are same. This can be achieved by adjusting one of the exponent up or down as required and dividing or multiplying the fraction by $r$ (shifting the fraction in the appropriate way) as many times as $e_1$ and $e_2$ differ. Once this is done, Case II reduces to Case I, and operation is carried out.

**Example 8.5** : Assuming 5 digit precision for fraction

add $X_1 = \cdot 23500 \times 10^9$ and

$X_2 = \cdot 45211 \times 10^7$

We can carry out the addition of $X_1$ and $X_2$ in two ways, *i.e.*, by picking (*i*) $e_1$ for adjustment or (*ii*) by picking $e_2$ for adjustment.

**(*a*) Picking $e_1$ for the Adjustment**

Decrease $e_1$ by 2 and multiply the fraction $f_1$ by 100 (shift $f_1$ left by two places). In this process, we shall loose two most significant digits of $f_1$. This gives an OVF in

fraction adjustment thus giving incorrect result.

**(b) Picking $e_3$ for Adjustment**

Increase $e_3$ by 2 to make it 9 *i.e.*, equal to $e_1$. Divide the fraction $f_3$ by 100 (shift $f_3$ right by two places). In this process, we loose two least significant digits of $f_3$. This is better than loosing most significant digits as in (*a*). The result of the addition is shown below :

$$X_1 = 0{\cdot}23500 \times 10^9$$
$$X_3 = 0{\cdot}45211 \times 10^7$$

$X_3$ with adjusted exponent is $0{\cdot}00452.\ 10^9$

$$X_1 = 0{\cdot}23500 \times 10^9$$
$$+X_3 = 0\ 00452 \times 10^9$$
$$S = 0{\cdot}23952 \times 10^9$$

From the above discussion it is clear that the exponent adjustment should be done on a number with smaller exponent. Exponent adjustment requires the division of fraction by $r$ as many times as two exponents differ. This thus keeps the value of the adjusted floating point number approximately (due to the constraints of available precision) as that of the original number. It should be clear now that *why $r^e$ is used as the scale factor* and not any other form like $k^e$ ($k \neq r$).

### 8.3.2. Floating Point Multiplication and Division

Multiplication and division operations on floating point numbers can be carried out as follows :

Let $X_1 = f_1.r^{e_1}$ and

$X_3 = f_3.r^{e_3}$ be two numbers.

Their product is obtained by simply multiplying the fractions and adding the exponents.

*i.e.*, Let $X_1.X_3 = P = f_p.r^{e_f}$

where $f_p = f_1.f_3$ and

$e_p = e_1 + e_3$.

The floating point division $X_1/X_3$ is carried out by dividing the fractions and subtracting the exponents.

$Q = X_1/X_3 = f_q.r^{e_q}$ where

$f_q = f_1/f_3$ and $e_q = e_1 - e_3$

### 8.3.3. Normalised Floating Point Representation

An unsigned floating point number is said to be in the normalised form if the most significant digit of its fractional part is non-zero. For example, consider $X = 012 \times 10^8$ and $Y = {\cdot}231 \times 10^8$. In these numbers X is not normalised, while Y is in normalised form. We can find out the normalised form of a number from its unnormalised form. For example X in the normalised form is ${\cdot}120 \times 10^8$.

When fractions are in sign—magnitude form, the definition of normalisation just given holds. On the other hand, if fractions are in sign-complement form, then a normalised number in a base $r$ has a most significant digit with the value not equal to $r-1$ for negative numbers.

**Example 8.6** : Let $N_1 = +{\cdot}012 \times 10^5$

$N_3 = -{\cdot}002 \times 10^4$

In the sign—10's complement representation these are :

$$N_1 = 0{\cdot}012 \times 10^5$$
$$N_3 = 1{\cdot}998 \times 10^4$$

Normalised numbers are :

$$N_1 = 0{\cdot}120 \times 10^4$$
$$N_3 = 1\ 800 \times 10^3$$

It is helpful if we keep the numbers in the normalised form. If all the numbers are stored in normalised form, the ALU can assume them in the normalised form and simply carry out the required operation and produce result in the normalised form. Following example illustrates the addition of normalised and unnormalised numbers.

**Example 8.7** : Add $A = 0{\cdot}00122 \times 10^3$ to $B = 0{\cdot}00011 \times 10^5$.

Since A has a smaller exponent, it will be picked up for adjustment and its fractional part shifted to the right.

Thus, $A = 0{\cdot}00001 \times 10^5$

$$B = 0{\cdot}00011 \times 10^5$$
$$S = 0{\cdot}00012 \times 10^5$$

If A and B are originally in the normalised form, we have :

$$A = 0{\cdot}12200 \times 10^{3}$$
$$B = 0{\cdot}11000 \times 10^{3}$$

After exponent adjustment A becomes $0{\cdot}01220 \times 10^{3}$.

Thus the sum is :

$$A = 0{\cdot}01220 \times 10^{3}$$
$$B = 0{\cdot}11000 \times 10^{3}$$
$$S = 0{\cdot}12220 \times 10^{3}$$

The result obtained by using the normalised representation for the above example is more accurate than that using unnormalised numbers. This clearly illustrates the importance of normalisation in floating point arithmetic. Normalisation of a floating point number is carried out by shifting the fraction to the left till a non-zero digit [or a non-$(r-1)$ digit in case of negative numbers in sign-r's complemen form] takes position at most significant digit position. Also, for every left shift on the fraction, the exponent is decreased by one. The discussion presented here on floating point arithmetic used decimal system for easy understanding of the principles, and examples of binary floating point arithmetic operations should be worked out by readers as an exercise.

## 8.4. FLOATING POINT ALGORITHMS

Based on the principles discussed in Sections 8.2 and 8.3, we now present algorithms to carry out the floating point operations. It is assumed (in the following algorithms) that the mantissae of floating point numbers are in sign-complement form, while the exponents are not assumed in any particular form. Exponent arithmetic should be done as per the actual representation.

Let X and Y be the floating point numbers having $e_x$ and $e_y$ as exponents and $f_x$ and $f_y$ as the mantissae respectively. Let $m_x$, $m_y$ and $m_s$ be the magnitude parts and $s_x$, $s_y$ and $s_s$ be the signs of mantissae $f_x$, $f_y$ and $f_s$ respectively as shown.

| | mantissae | | exponents |
|---|---|---|---|
| Operand X : | $s_x$ | $m_x$ | $e_x$ |
| Operand Y : | $s_y$ | $m_y$ | $e_y$ |
| Result S : | $s_s$ | $m_s$ | $e_s$ |

The algorithms presented here are for numbers in any general radix $r$, and the shift operations involved in these algorithms are arithmetic shifts.

### 8.4.1 Addition/Subtraction

**Algorithm 8.1. Floating Point Addition/Subtraction**

**Steps**

1. If operation is subtraction then complement $f_y$.

2. If $e_x > e_y$ then exchange X and Y.

**Comment** : After this step X has a smaller exponent.

3. If $e_x = e_y$ then go to step 6.

4. Shift $f_x$ right by 1 digit position *i.e.*, divide $f_x$ by $r$, and increment $e_x$ by 1.

5. Go to step 3.

6. CY $\leftarrow 0$, $f_s \leftarrow f_x + f_y$, $e_s \leftarrow e_y$

**Comment** : CY is a carry F/F which stores carry resulted in the addition from the sign bit of resultant mantissa.

7. OVF $= \overline{x_s \oplus s_y} \cdot (s_s \oplus s_y)$

8. If OVF=1 then shift $f_s$ right by one digit. $s_s \leftarrow$ CY and increment $e_s$.

9. If $e_s$ overflows then set exponent OVF (EOVF), and go to step 13.

10. If $s_s$ + MSD of $m_s$ (Most Significant Digit of $m_s$) $\neq r$ or 0 then go to step 13.

11. Shift $f_s$ left by one place and decrement $e_s$.

12. If $e_s$ underflows (borrow) then set exponent underflow (EUDF), and go to step 13, otherwise go to step 10.

13. Stop.

**Comment** : Steps 10, 11, and 12 carry out normalisation.

### 8.4.2. Multiplication

**Algorithm 8.2. Floating Point Multiplication Steps**

1. $f_z \leftarrow f_x \cdot f_y$
2. $e_z \leftarrow e_x + e_y$
3. If $e_z$ overflows (or underflows) then set exponent OVF (EOVF) (or set exponent underflow (EUDF) and go to step 7.
4. If $s_z$ + MSD of $m_z$ (Most Significant Digit of $m_z$) $\neq r$ or 0 then go to step 7.
5. Shift $f_z$ left by 1 position and decrement $e_z$.
6. If $e_z$ underflow (borrow) then set exponent underflow (EUDF) and go to step 7, otherwise go to step 4.
7. Stop

**Comments.** Steps 4, 5 and 6 carry out the normalisation of the product.

### 8.4.3. Division

**Algorithm 8.3. Floating Point Division**

This algorithm have same steps as those in Algorithm 8.2 except the first two steps. These two steps are as follows :

1. $m_z \leftarrow m_x / m_y$
2. $e_z \leftarrow e_x - e_y$

## 8.5. EXPONENT REPRESENTATIONS AND THEIR EFFECTS ON LOGIC CIRCUIT IMPLEMENTATION

From the algorithms presented in section 8.4, it is to be noted that exponent adjustment and/or normalisation operations are invariably present in these algorithms. Thus, an exponent representation better suited to the adjustment operation should be selected for implementation of floating point algorithms.

### 8.5.1. Exponent Adjustment

In this section we shall study the exponent adjustment process at logic circuit level, when exponents are in various representations

Let X and Y are two floating point numbers with exponents $e_x$ and $e_y$ respectively in sign—magnitude form. Assume that $e_x$ is smaller than $e_y$. If $e_x$ is negative the register which contains the magnitude of $e_x$ is to be decremented and at the same time $m_x$ (mantissa of X) is shifted right. If $e_y$ is also negative, the adjustment is terminated when $e_x$ reaches $e_y$. On the other hand if $e_y$ is positive, the decrementation of magnitude of $e_x$ shall continue till a—0 is reached and then the sign of $e_x$ has to be complemented and further adjustment continued and decrementation of magnitude of $e_x$ has to be changed to incrementation. At every step of the adjustment, a comparison between $e_x$ and $e_y$ is made and the adjustment stopped when they match.

When exponents are represented in sign—complement form, the steps are much simpler than those discussed for sign—magnitude form. In the present case, the smaller exponent is incremented (treating it as an unsigned number) and fraction of the smaller number is shifted) till the two exponents match. Exactly similar situation exists for biased exponents. From this discussion, it may be noted that exponents in sign—magnitude form require (*i*) up/down counters for exponent adjustment and (*ii*) comparator to compare two signed integers in sign—magnitude form, on the other hand when exponents are in sign—complement form, only up counter and comparator for comparing numbers in sign—complement form is required. Exponents in biasd form require only up counter and a comparator for comparing two unsigned integers. Obviously, biased exponents should be first on merit as far as exponent adjustment is concerned, leaving exponents in sign—2's complement notation in the second place.

**Example 8.8.** This example illustrates the discussion of section 8.4 for decimal floating point numbers.

Consider the two floating point numbers with exponents $e_x = -3$ and $e_y = +2$. The exponent $e_x$ is picked up for the adjustment and the adjustment process is as follows for the three representations.

| *Sign—Magnitude* | *Sign—10's Complement* | *Biased (Excess 5)* |
|---|---|---|
| *Initial Condition* $e_x = -3$ | *Initial Condition* $e_x = 1,7$ | *Initial Condition* $e_x = 2$ |
| *Steps* | *Steps* | *Steps* |
| 1. Decrement $\lvert e_x \rvert$ : $e_x = -2$ | 1. Increment $e_x$ : $e_x = 1, 8$ | 1. Increment $e_x$ : $e_x = 3$ |
| 2. Decrement $\lvert e_x \rvert$ : $e_x = -1$ | 2. Increment $e_x$ : $e_x = 1, 9$ | 2. Increment $e_x$ : $e_x = 4$ |
| 3. Decrement $\lvert e_x \rvert$ : $e_x = -0$ | 3. Increment $e_x$ : $e_x = 0, 0$ | 3. Increment $e_x$ : $e_x = 5$ |
| 4. Complement sign of $e_x$ : $e_x = +0$ | 4 Increment $e_x$ : $e_x = 0, 1$ | 4. Increment $e_x$ : $e_x = 6$ |
| 5. Increment $\lvert e_x \rvert$ : $e_x = +1$ | 5. Increment $e_x$ : $e_x = 0, 2$ | 5. Increment $e_x$ : $e_x = 7$ |
| 6. Increment $\lvert e_x \rvert$ : $e_x = +2$ | | |

## EXERCISES

**1.** A 10-bit floating point word has 6 bits for mantissa (including sign) in sign—magnitude form and 4 bits for exponent (with base 2) in excess 8 form.

(*a*) Represent $X = 13{\cdot}5)_{10}$ and $Y = -9\ 25)_{10}$ as the normalised floating point binary numbers in the above format.

(*b*) Illustrate the floating point addition of X and Y.

(*c*) Illustrate the floating point division of X and Y.

(*d*) Illustrate the floating point multiplication of X and Y.

2. Repeat (1) for the numbers $X = -3{\cdot}25$ and $Y = +8.5$.

3. Repeat (1) when mantissa is in sign—2's complement form.

4. Find the range of floating point numbers that could be represented in the floating point format of (1).

5. Interpret the following hexadecimal numbers as floating point numbers in the IBM 360 floating point format (single precision) and write their closest decimal floating point equivalents.

(*a*) 44024E00

(*b*) 34618256

(*c*) 69AB1F12
(*d*) 151ABCD10

6. Repeat (5) for HP2100A minicomputer.

7 Find the range of numbers that could be represented in the IBM360 single precision floating point format.

8. Repeat (7) for HP2100A.

9. Write the floating point addition/subtraction, multiplication and division routines in the machine language of a machine you have.

If you do not have a computer, decide on the format of floating point numbers for the hypothetical machine done in Chapter 1, and code them in the machine language of the hypothetical machine.

□

# 9

# CORE MEMORY SYSTEMS

Instructions and data on which a central processing unit works are stored in the main memory. It is a major component of a digital computer system. The size of such a memory ranges from a few kilo bytes in small computers to a few million bytes in large computer systems. In early systems, magnetic cores were the only form of memories used as main memory and the terms core memory and main memory were used synonymously. With rapid advances in semiconductor technologies from early seventies, semiconductor memories are replacing core memories. Inspite of this development, core memories continue to be main memories in many systems. In this chapter, we shall study the principles of core storage and their organisations in forming large scale memory systems.

## 9.1. MAGNETIC CORE

A magnetic core is a small, toroid shaped ring. Wires can be strung through it for passing electric current (Fig. 9.1). The B-H curve, *i.e.*, a graph between the applied magnetising force H (which is proportional to the applied current I) and the resultant flux density B is shown in Fig. 9.2 for a typical magnetic core. When an electric current of I amperes is passed through the wire, it induces a magnetic force $H \propto I/R$ at radius R (Fig. 9.1). The magnetising force H induces a flux density B in the core in the direction of H, which in turn depends on the direction of the current I. The relation between B and H is determined by the material used for making cores. Ferrite cores used for memory systems have a B-H curve as shown in Fig. 9.2.


FIG. 9-1 A MAGNETIC CORE

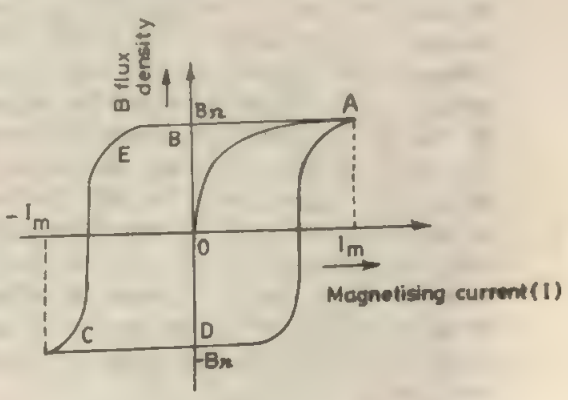



FIG. 9-2 B-H CURVE FOR FERRITE CORE

## 9.2. CORE AS A STORAGE CELL

Let us consider an unmagnetised core *i.e.*, core with the initial condition corresponding

to the origin on the BH plane. Let the current I be increased from 0 onwards gradually in the positive direction (which is arbitrarily chosen). For this change of I, the flux density moves along the line OA. Any further increase of current (*i.e.*, from current value $I_m$) from the point A does not significantly change the flux density and the core is said to be saturated in the positive direction. Now if the current I is decreased towards zero, magnetisation does not follow the line AO, but follows line AB (Fig. 9.2) due to the phenomenon called hysteresis. Thus at the point B, which represents I=0, the core has a flux density of magnitude $+B_r$, which is called the residual flux. If we continue to decrease I beyond zero (*i.e.*, pass current in the negative direction), the B—H curve follows the path BEC and at the point C, the core saturates in the negative direction. The release of the current to zero value now leaves the core with the residual flux density of $-B_r$ at the point D on the B—H curve. The shape of the hysterisis curve (B—H curve) of a magnetic core is such that, an already saturated core (in either direction) does not appreciably change its flux density even for the currents of values upto 6 $I_m$. From the behaviour of the core discussed above, we give the following important conclusions.

(*a*) We can make a core to retain either of the flux densities $+B_r$ or $-B_r$ by passing a current of value $+I_m$ or $-I_m$ (see Fig. 9.2) respectively ($I_m$ will be termed as full saturation current or simply full current).

(*b*) It the core has to retain maximum magnetisation, the B—H curve should be rectangular.

(*c*) $\pm I_m/2$ current does not cause any appreciable change in the residual flux of a saturated core. This value of current shall be termed as half current.

(*d*) A core can retain (store) either $+B_r$ or $-B_r$ flux density depending on how it was saturated. These could be used as two logical states say $+B_r$ as logical 1 and $-B_r$ as a logical 0.

(*e*) A core can be used as a storage cell capable of storing one bit of information.

### 9.2.1. Write operation

As discussed earlier, we record a 1 in the core cell by passing a current $I_m$. To record a 0, a current of magnitude $-I_m$ is passed. The duration for which these currents are passed should be sufficient to saturate the core This duration is a property of the magnetic material and the core geometry. It is of the order of half a microsecond. It is called the core switching time. Note that $\pm I_m$ current could be passed by a single wire carrying a full current or two separate wires each carrying half current in the additive direction. These wires are called drive wires. There may be two or three drive wires for a single core.

### 9.2.2. Read operation

The information contents of a core cell cannot be directly read. To read a core cell we need a special read electronics It consists of a read wire passing through the core and a sense amplifier, which amplifies and shapes the voltage on the sense wire into logical state voltage. The read procedure is as follows. We pass a current of value $-I_m$ through the drive winding ; this takes the core into a logical 0 state. If it were storing a logical 1, there would be a change of flux corresponding to the difference between the residual flux densities (*i.e.*, $+B_r-(-B_r)=2B_r$). This change produces a large voltage on the sense winding and the sense amplifier gives out a logical 1 signal for some fixed duration. On the other hand, if the core were storing a 0, negligible voltage appears on the sense winding and a logical 0 signal is given by the sense amplifier. Note that when a core is read it always stores a 0 after read operation, thus destroying the old contents. Hence the core read operation is destructive.

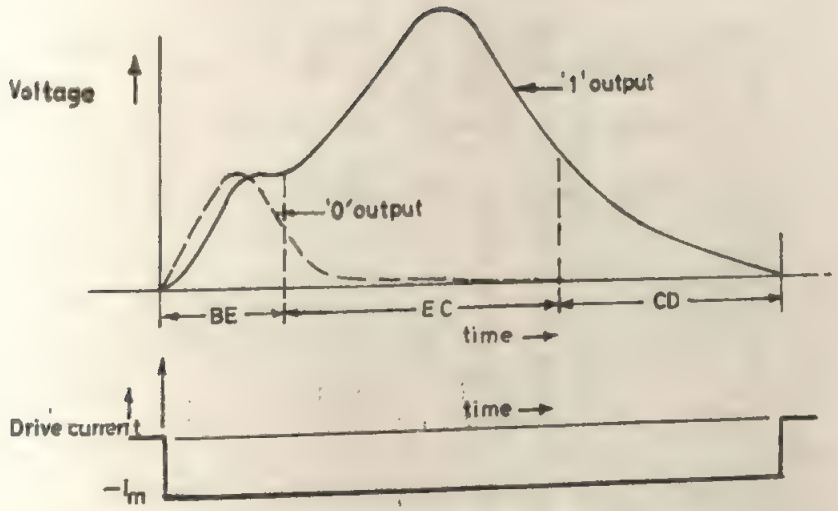



FIG.9.3 VOLTAGE INDUCED ON SENSE WIRE

Moreover, sense amplifier output signal is available only for a small duration, and therefore it must be latched into a flip-flop.

The typical output voltage wave forms on the sense winding are shown in Fig. 9.3. Qualitatively these can be explained as follows. When the core is storing a 1 the read current $-I_m$, takes the core from the point B to D through the path BECD (Refer Fig. 9.2). There is a small change in flux (due to non-zero slope of EB) from B to E; this induces a small voltage peak, after which the flux changes from point E to C which accounts for a major flux change and consequently a large voltage is induced in the sense winding. This voltage decreases to zero in the region CD. The waveform of the induced voltage for a core storing a 0 is shown with a dashed line. In this case a very small flux change occurs in the region DC, which gives a small peak, after which voltage decays towards 0. The 1 and 0 waveforms on the sense winding differ not only in amplitudes but also in the times at which the peaks occur. Therefore, they can be easily distinguished

The following sections deal with various organisations of basic core cells and drive electronics in forming a memory system. A memory unit is basically organised as an array of number of words (locations), where each word contains a number of bits. The word is a basic unit of information which could be transferred in and out of the memory in one access. The size of the memory is usually expressed in terms of *n* words of k bits each (*i.e.*, $n \times k$).

## 9.3. LINEAR SELECTION (2D) ORGANISATION

To bring out the salient features of this memory organisation, we shall consider a $64 \times 4$ 2D memory, whose schematic organisation is shown in Fig. 9.4

### 9.3.1 Organisation

In a $64 \times 4$, 2D memory organisation, the cores are arranged in the form of a matrix of size $64 \times 4$ (Fig. 9.4.) ($n \times k$ for a memory of size $n \times k$). A row of the matrix corresponds to a word of the memory. The cores in a row have a common drive wire called X drive winding. While the cores in

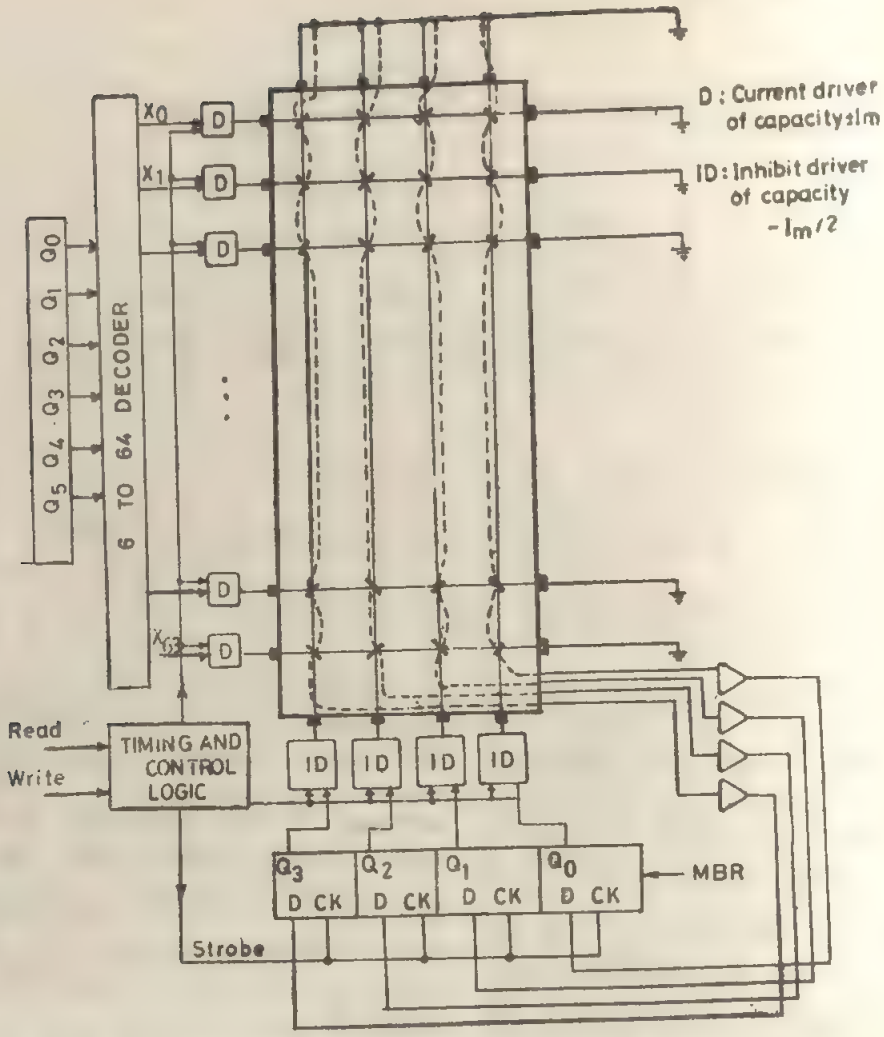



FIG.9·4 64X4 2D MEMORY ORGANISATION

a column have a common Y drive (bit drive) winding and also a separate sense winding as shown in Fig. 9.4. X wires $X_0, X_1 \ldots, X_{63}$ are driven by current drivers $D_0, D_1 \ldots, D_{63}$ respectively. The X current driver, when enabled can drive current of $\pm I_m$ amperes. On the other hand, Y wires are driven by inhibit drivers (IDs). An inhibit driver, when enabled, drives $-I_m/2$ amperes. The 6 to 64 decoder selects one of the 64 X drivers for sending the X drive current through the cores of a required location. An inhibit driver is enabled when there is 0 in the corresponding bit of MBR. The sense winding output voltage is amplified and shaped into 0 or 1 voltage by the sense amplifiers.

### 9.3.2. Read cycle

The read cycle of this memory consists of passing a current of value $-I_m$ through the selected X wire (*i.e.*, cores of a selected location). A data bit appearing on a sense wire is due to the flux change in the selected core. Outputs of the sense amplifiers form a data word read from the addressed location. The sense amplifiers outputs are latched into MBR

by the strobe pulse. At the end of read cycle, it is to be noted that the addressed location shall contain all 0's.

### 9.3.3. Write cycle

Assuming that all the cores of a selected location contain 0's, we shall discuss, how a data word could be recorded in any required location. The addressed location is selected by the decoder which enables the appropriate current driver. This current driver passes the current of value $I_m$ through the selected X wire (*i.e.*, through all the cores of selected location). If this is the only drive current, all the selected cores will switch to a logical 1 state and thus all 1's shall be recorded in the addressed location, irrespective of the data pattern to be recorded. To avoid this the inhibit windings are provided. An inhibit wire is driven by an inhibit Driver (ID) which drives a current of value $-I_m/2$, *only when the corresponding MBR bit has a 0.* This $-I_m/2$ amperes current and an X current of value $+I_m$ amperes add to $+I_m/2$ for the selected core in which a 0 is to be recorded. The value $+I_m/2$ amperes is not enough to switch the core and thus it remains in its old state (*i.e.*, 0). If a 1 is to be recorded, the inhibit driver is not enabled, and the selected core switches to a 1 state.

### 9.3.4. Memory Cycle

Any request to the memory for fetching or storing a data word is internally mapped

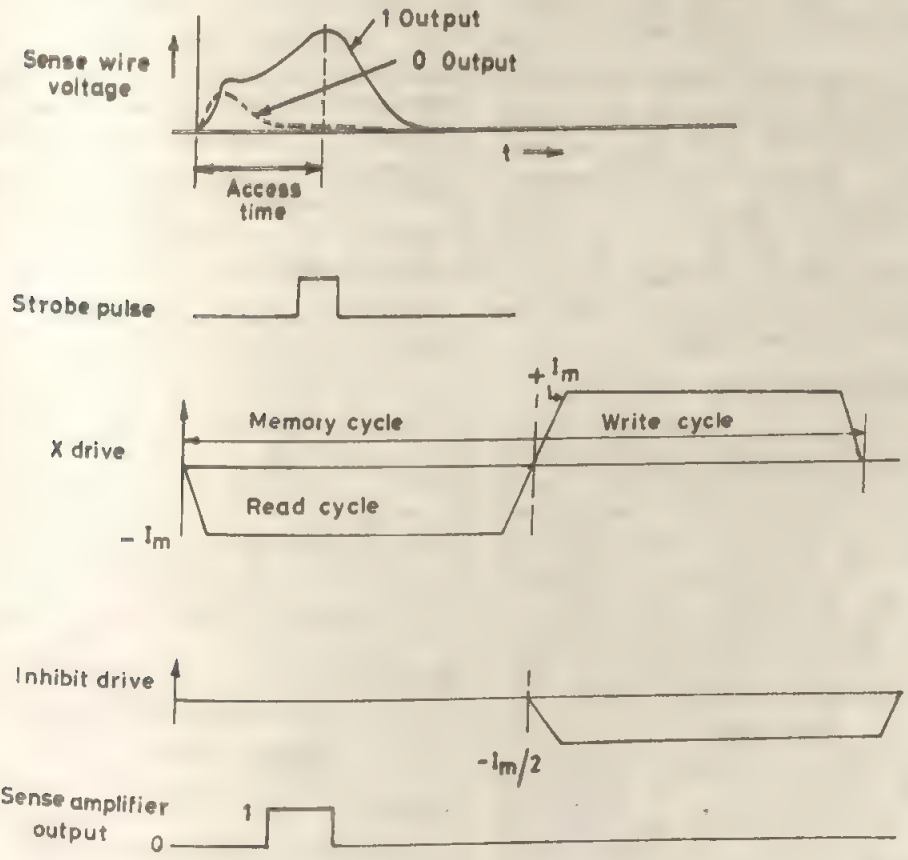



FIG 9.5 3D MEMORY CYCLE AND CURRENT WAVE FORMS

by the memory controller into a memory cycle. A memory cycle consists of a read cycle and write cycle. In read cycle, data is read and the location is cleared due to destructive read operation and therefore write cycle follows the read cycle, which records the data read in the read cycle (or a data supplied in MBR).

When a request for storing a data word is made, memory controller generates a read cycle to clear the addressed location (since write cycle requires 0's to be present initially in cores). The data generated on the sense wires during this read cycle is prohibited from entering into the MBR. The write cycle then records the given data (which should be present in the MBR before the write cycle commences) in the addressed location.

Fig. 9.5 shows the 2D memory cycle current waveforms with important parameters like access time and cycle time marked. Access time is the time required by memory to make available the data from a required location, in MBR. As can be seen from Fig. 9.5, access time is less than read cycle time, while memory cycle time is twice the time required to switch a core.

### 9.3.5. Discussion

It can be easily seen from Fig. 9.4 that a 2D memory of $n \times k$ size shall require $n+k$ current drivers. Therefore for a large value of $n$, the decode and drive electronics is too expensive for this organisation to be practicable. On the other hand, induced emf in each sense wire is entirely due to the core which is being read, (this is not the case with other organisations) and thus there is no limitation on the number of cores (*i.e.*, locations) on one sense winding. Also, since inhibit wire is required only in write cycle and sense wire is required during the read cycle, a single wire could do both these functions. This single wire can act as a sense winding during a read cycle and as an inhibit winding during a write cycle. Moreover, in a read cycle, cores of only the selected location are affected. This has the advantage that by increasing the read current $-I_m$, we can get better access time.

## 9.4. COINCIDENT CURRENT (3D) ORGANISATION

In this section, we shall discuss 3D memory organisation also called as coincident current memory organisation

### 9.4.1. Organisation

As the name suggests, we have a core stack arranged in three dimension of space (at least conceptually if not physically). A 3D memory of size $n \times k$ has $k$ planes of cores. One plane contains $n$ cores, each core representing *a bit of one location. Thus, one plane forms one bit of all the locations.* These $k$ planes are stacked to form the $k$-bit per word memory. In this section we shall discuss a $64 \times 4$, 3D memory to bring out the general principles. A typical plane of 3D memory is shown in Fig. 9.6.

Each of these planes has 8 rows ($\sqrt{n}$ in a general case) and 8 columns. Through each row of cores passes a drive wire called X drive wire. Let these wires be numbered as $X_0$, $X_1$..., $X_7$. Similarly, through each column passes a Y drive wire. Let these also be numbered as $Y_0$, $Y_1$..., $Y_7$. There is one sense winding passing through all the cores in the plane and its geometry is shown in Fig. 9.6. The reason for this geometry shall be discussed later. A plane also has an inhibit winding (not shown in Fig. 9.6) passing through all the cores, such that inhibit current is in the opposite polarity to that of X and Y drive currents. Such a 3D organisation is called 3D—4 wire organisation. If a common winding is used for sense/inhibit purpose, the organisation is called 3D—3 wire organisation. A 3D—3 wire organisation is slightly complex, because sense and inhibit winding

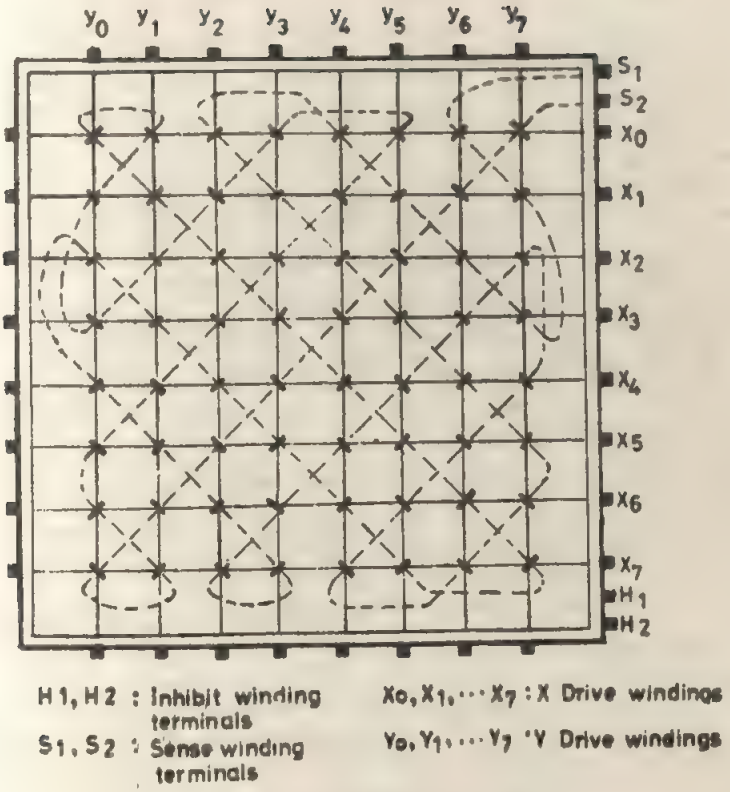



FIG. 9·6 3D MEMORY CORE PLANE

geometries are decided by different factors. Though the geometries of the sense and inhibit functions are conflicting, special electronics could be used to alter the direction of inhibit current so that it shall oppose the X or Y drive current for all cores.

We shall discuss, in this section, a 3D—4 wire organisation, nevertheless the concepts discussed in this section also apply to 3D—3 wire organisation.

The block diagram of a 64×4, 3D memory organisation is shown in Fig. 9.7. As can be seen from Fig. 9.7, the X drive wires of the planes are connected in series and finally grounded. Similar connections are made for Y drive wires. The bits of Memory Address Register (MAR) are divided into two parts. Each part has half the total number of bits. One part addresses the X drivers, while the other addresses the Y drivers. The X and Y parts of MAR, which are decoded separately, enable the addressed X and Y drivers. For selecting a core, a half select current is passed through the selected X and Y wires through their respective drivers. This produces a magnetising forces equivalent to a full current for the *core at the intersection of the X and Y lines*. This is the required core addressed by the MAR. One such core exists on each plane and these cores form a memory word for the addressed location. Also in each plane, other cores on the selected row and column get half select current. As discussed earlier in Section 9.2, these cores are not affected. The sense winding of a plane feeds the sense amplifier through a rectifier. An inhibit driver of a plane is enabled in the write cycle to send a current of $-I_m/2$ if the corresponding bit in MBR is 0, otherwise it is not enabled.

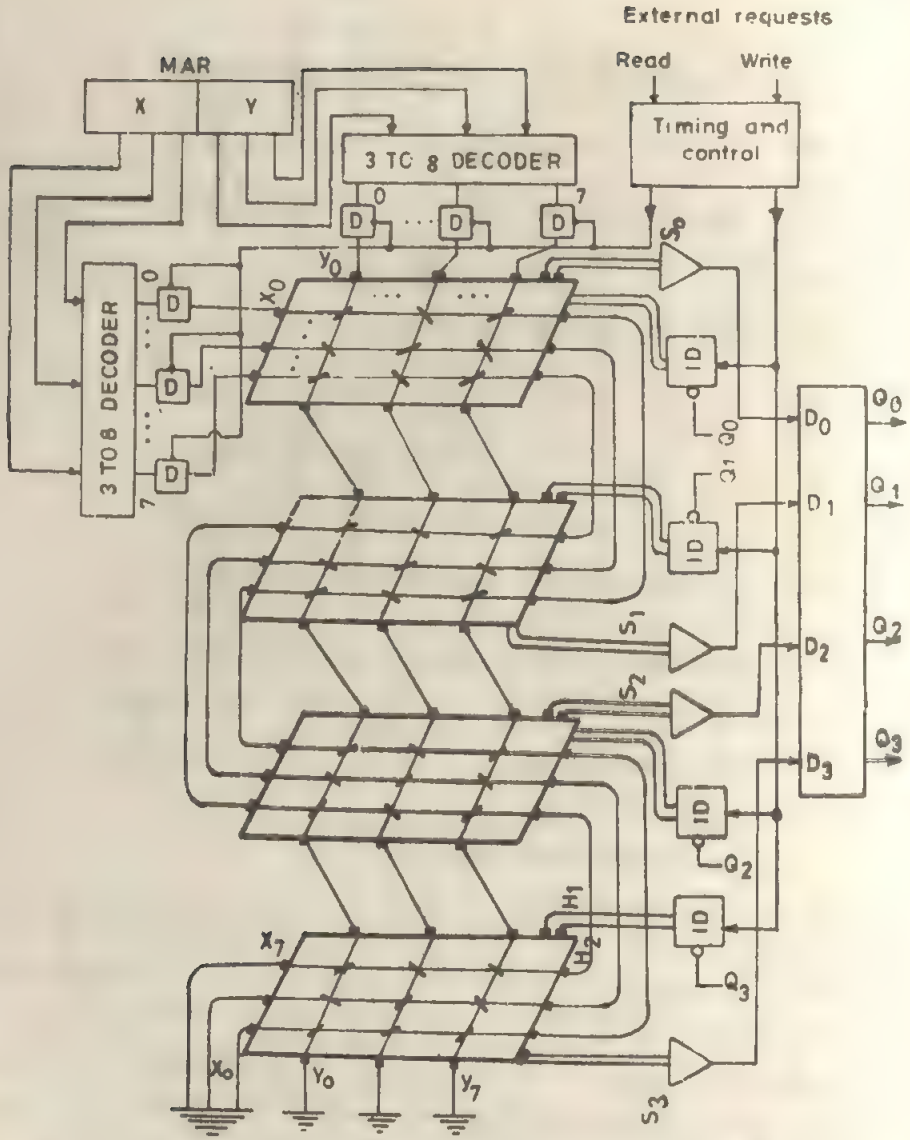



FIG. 9·7 64X4 3D CORE MEMORY ORGANISATION

### 9.4.2. Read Cycle

In a read cycle for a 3D memory, a current of $-I_m/2$ is passed through the selected X and Y drive wires by their respective drivers. The selected core in each plane gets a full current $-I_m$ and switches them to 0 state. This switching induces emfs on the sense windings of planes, thus giving the state of the selected cores (before switching) at the output of the sense amplifiers. The data at outputs of sense amplifiers is stored in the MBR. It may be observed that each plane also has $N_x+N_y-2$ ($N_x$, $N_y$ denote the number of X and Y drive wires in a plane) half select cores. These half selected cores induce emfs similar to a logical

0 output voltage on the sense winding. These voltages are small and their peaks do not appear at the strobe time. But inspite of their small values at the strobe time, they may jointly contribute to a large voltage, which could possibly be falsely interpreted as a logical 1. Due to this reason, sense winding is threaded such that a pair of cores in a row or column cancel the half select emfs of each other One such threading is shown in Fig. 9.6. Due to this particular threading, the selected cores of various locations will not produce emfs of the same polarity, therefore the sense winding output must be rectified and then applied to the sense amplifiers.

### 9.4.3. Write Cycle

In the write cycle, a current of value $+I_m/2$ is passed through the selected X and Y wires and a current of value $-I_m/2$ is passed through the inhibit windings of bit planes where the corresponding MBR bits are 0's. In the absence of the inhibit currents, the selected cores in all the planes will get a full current and thus would switch to the logical 1 state for all the planes, thus recording a pattern of all 1's in the addressed location. To record a 0 in a particular bit, an inhibit current of the magnitude $-I_m/2$ is passed by the corresponding inhibit driver, which reduces the total current of the selected core to $+I_m/2$. This, thus leaves the selected core in its old state (which would be 0 due to the read cycle carried out earlier).

### 9.4.4. Memory Cycle

Memory cycle for a 3D core memory

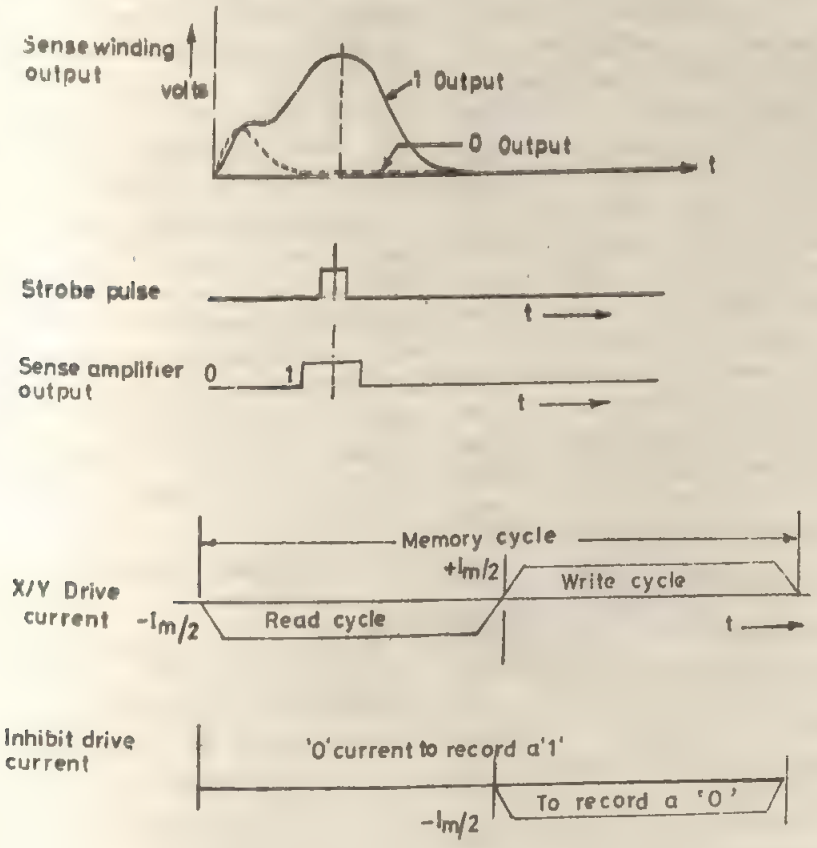



FIG. 9·8 3D MEMORY CYCLE AND WAVEFORMS

constitutes a read cycle followed by a write cycle. When a data word is to be fetched from a location, memory unit carries out a read cycle. This brings the data into the MBR and leaves the addressed location in the zero state (due to destructive read out). To restore the data just read, a write cycle is carried out. Similarly for recording a data word, a read cycle clears the addressed location and a write cycle following it puts the data (which should be in the MBR before the write cycle starts) in the addressed location. Fig. 9 8 shows the current and related waveforms for a 3D memory cycle.

### 9.4.5. Discussion

A 3D core memory organisation offers cost—effectiveness in the drive electronics (*i.e.*, number of drivers/decoders). On the other hand, due to a large number of half selected cores ($N_x+N_y-2$) the noise on sense windings has to be taken care of. This is done, as mentioned earlier, by a peculiar threading of the sense windings. A 3D—4 wire organisation is thus simple and inexpensive (except for sense amplifiers which require higher signal to noise ratio). On the other hand 3D—3 wire organisation is some what complex, since for every core in a plane the inhibit current has to be in the opposite direction to that of X and Y drive currents. This is a complex task if sense winding is used as inhibit winding, because sense winding direction alternates for every consecutive cores in a row or column. Therefore to keep the inhibit current in the required direction, a special electronics has to be used, which would switch the direction of the inhibit current appropriately depending upon the position of a selected core with respect to the sense winding. The 3-wire memories can be built with smaller size of cores than those with 4 wires, and smaller the core, less is its switching current and the time required for switching. This would result in faster memories. On the other hand we can not switch a core faster, by increasing the current beyond $I_m$, because, in 3D organisation, drive current cannot be increased to a great extent since the half selected cores would also get more current. This half current if sufficient, may switch these cores thus destroying the data they were storing.

## 9.5. 2½D MEMORY ORGANISATION

### 9.5.1. Organisation

A core plane for this organisation is arranged in the same manner as is done for a 3D organisation, with the exception that there is no inhibit winding. The plane is made of a rectangular matrix of size $m \times n$. The numbers $m$ and $n$ are selected such that the total number of drivers required are minimised. Fig. 9.9 shows the schematic of such an organisation for a $64 \times 4$, 2½D memory. For convenience, the planes are not shown one below the other. The X and Y wires pass half select currents when selected. Since there are separate Y drives for each plane, the selected Y driver of a plane may be separately enabled/disabled.

In this organisation, X wires are connected in series as is done in case of 3D memory. One end of X wires is grounded, while the other end is connected to a driver. The Y wires of individual planes have drivers which are terminated to ground separately. If X and Y denote the number of rows and columns in a plane, then, we require a total of X+K.Y current drivers for the K-bit word memory.

### 9.5.2. Optimum number of rows and columns

We shall give a simple method, to arrive at the organisation of a plane in terms of member of rows and columns.

Let X=Number of X wires (rows),
Y=Number of Y wires (columns),
K=Number of bits in a word,
N=Number of memory locations (*i.e.* number of cores in a plane) and
D=Number of drivers.

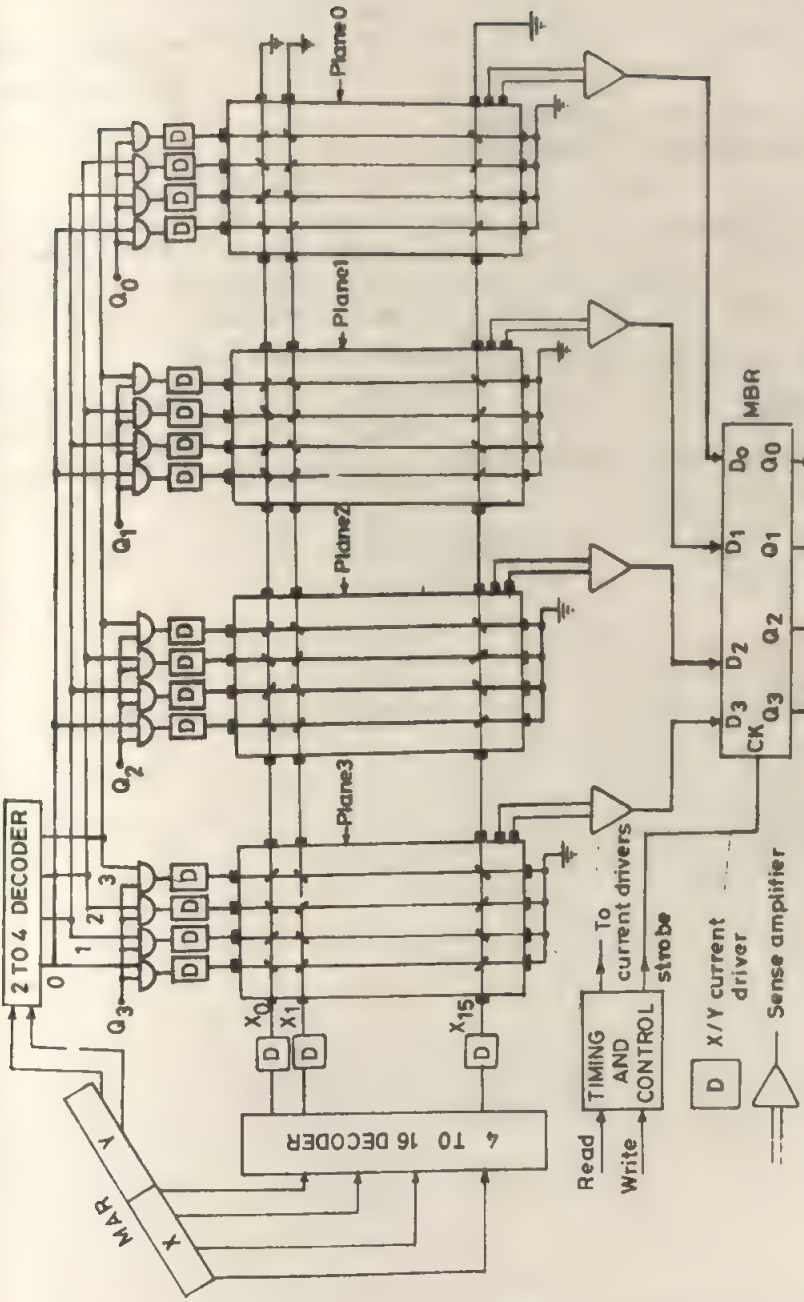



FIG. 9·9 64X4 $2\frac{1}{2}$ D CORE MEMORY ORGANISATION

Looking at Fig. 9.9, we can immediately write :

$$D = X + K \cdot Y$$

and $N = X \cdot Y$

thus $D = \frac{N}{Y} + K \cdot Y$ ...(9.1)

To obtain the minimum D, we differentiate Equation (9.1) with respect to Y and equate it to 0.

Thus $\frac{dD}{dY} = \frac{-N}{Y^2} + K = 0$

or

$$Y = \sqrt{N/K}$$

Since $\frac{d^2D}{dY^2}$ is −ve, we get a minima of D at

$$Y = \sqrt{N/K}.$$

From this we find the number of bits required in MAR to address the Y wires. This shall be $\log_2 Y$ (if $\log_2 Y$ is not a whole number, the nearest integer number may be taken).

For the 64 × 4 memory discussed, we have :

$$Y = \sqrt{\frac{64}{4}} = \sqrt{16} = 4,$$

and

$$X = \frac{N}{Y} = 16.$$

Thus out of 6 bits of MAR, 2 bits are used for addressing the Y wires, while 4 bits are used for addressing the X-wires.

### 9.5.3. Read Cycle

The selected X driver passes a current of value $-I_m/2$. This current passes through all the planes. Also, for each plane, the selected

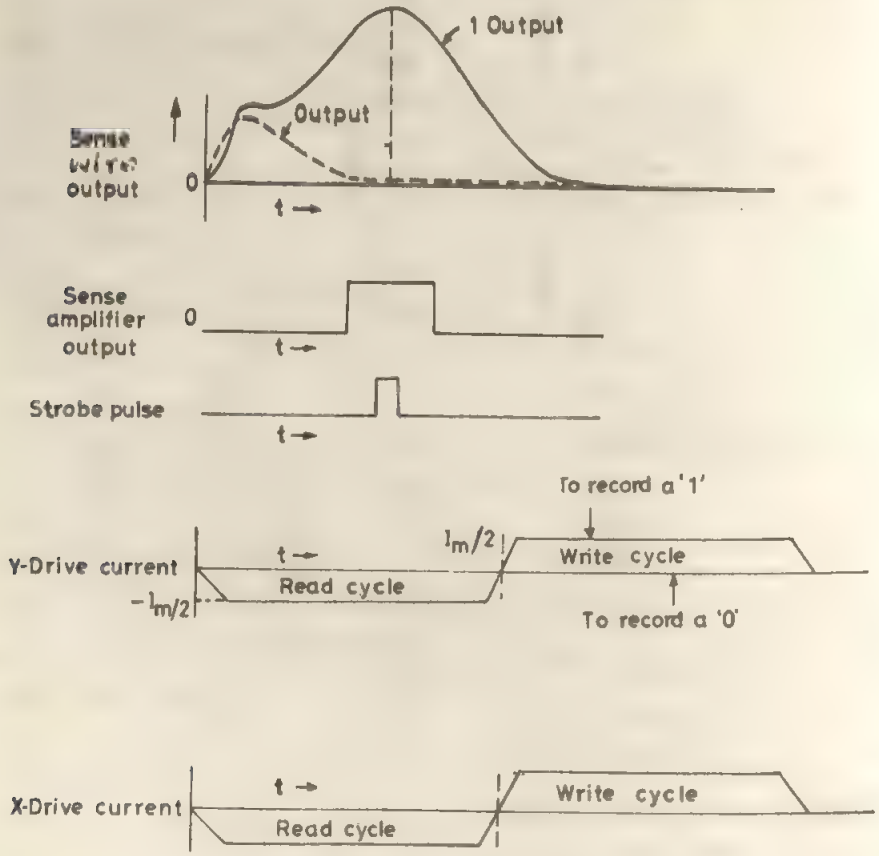



FIG. 9.10 $2\frac{1}{2}$D MEMORY CYCLE AND WAVEFORMS

Y driver passes the current of value $-I_m/2$, and the cores (one in each plane) *at the intersection of these X and Y* wires get full current. This causes these cores to switch to zero. The emfs, which are induced, are amplified and the data is strobed into MBR. The selected cores are left in 0 state.

### 9.5.4. Write Cycle

In a write cycle, the selected X driver passes $+I_m/2$ current. But the selected Y drivers pass $+I_m/2$ current only for the planes corresponding to MBR bits storing 1's. This, thus passes full current in these bit cores and switches them to 1. Other selected cores are left in the 0 state. For example, if a pattern of 1001 is to be recorded, then selected Y drivers of planes 0 and 3 are enabled to pass $+I_m/2$, while selected Y drivers for planes 1, 2 are not enabled. This gives full current $+I_m$ only to the selected cores in the planes 0 and 3, and therefore they are switched to 1. The selected cores in planes 2 and 3 are left in the old state (which is 0 due to the previous read cycle). The current waveforms for $2\frac{1}{2}D$ memory cycle are shown in Fig. 9.10.

### 9.5 5. Discussion

Due to separate Y drivers for each plane, $2\frac{1}{2}D$ organisation does not require inhibit winding. Since one less wire is to be threaded small cores could be used which give higher speed. The cost of drive electronics is higher for a $2\frac{1}{2}D$ memory than that for 3D organisation. In other respect, this organisation is similar to 3D organisation.

### 9.5.6. Comparison of Memory Organisation

In this section, we briefly summarise the various aspects of memory organisations. These are listed in table 9.1.

TABLE 9.1

**Comparison of Memory Organisations**

| *Sr. No.* | *Factor* | *Organisation* | | |
|---|---|---|---|---|
| | | *2D* | *3D* | *$2\frac{1}{2}D$* |
| 1. | Current drivers | N+K | $2\sqrt{N}+K$ | $X+K \cdot Y = 2\sqrt{N} \cdot K$ |
| 2. | Current effect on other cores during read cycle | Only selected cores are affected | Half current is passed through X+Y−2 cores | Same as 3D |
| 3. | Total number of wires through a core | 3 (can be easily reduced to 2) | 4 (could be reduced to 3) | 3 |
| 4. | Use | Small systems | Large systems | Large systems |
| 5. | Sense amplifier | Easy and less complex | Half select core noise has to be eliminated | Same as 3D |

## 9.6 SELECTION CIRCUITS USED IN PRACTICE

The arrangements discussed in the earlier sections for the selections of X and Y drive windings are never used in practice, because it is still possible to reduce considerably the drive electronics. For example, in a 4K, 3D core memory we shall require 64 X and 64 Y drivers. The number of drivers required for this memory using the technique discussed in this section shall only be 32 This reduction is possible using diodes and one further level of coincident selection on X and Y drive wires. We call this technique as 'coincident—coincident' selection. The basic principles involved in this technique can best be illustrated through an example. We discuss thus a 3D memory with 256 locations.

A 3D memory with 256 cores shall have 16 X and 16 Y drive windings. Since the principles involved in X and Y selection are same, the present discussion is confined only to the X selection. Similar circuits are to be used for Y selection. The X part of MAR (4 bits in the present case) is split into two equal subparts each subpart is 2 bits in length. Let XMS and XLS be the most significant and least significant parts respectively in the address of the X line. The 16 X drive wires are arranged in four groups, each group having 4 wires. Let a group be selected by the XMS part and a wire within a group be selected by the XLS part. The selection circuit is shown in Fig. 9.11. In this circuit the selection of one of the 4 drivers is done by the XMS part, while the selection of one of the 4 switches is done by XLS part. Consider for example the selection of the 14th wire (1110 as binary address). This wire is selected by the driver $D_3$ and switch $SW_2$ (Fig. 9.11). The driver $D_3$ sends the required current which can flow through one of the wires in the column of $D_3$. But since only $SW_2$ switch is ON, the current flows through the wire 14. Diodes shown at the end of a wires prevent the current in the selected X wire from being diverted to other wires in the row. A switch used in Fig. 9.11 has three positions, N (sink negative current), P (sink positive current) and 0 (open)

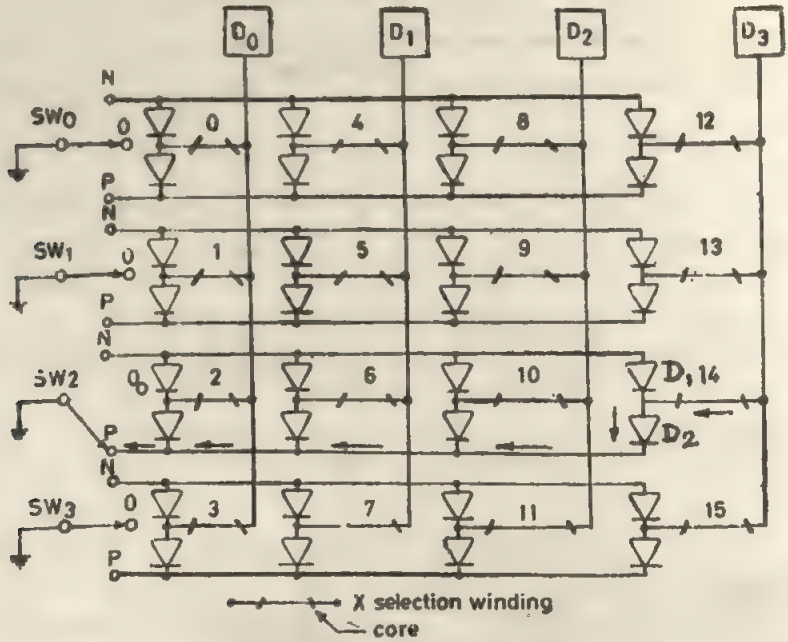



FIG.9·11 X-LINE SELECTION CIRCUIT

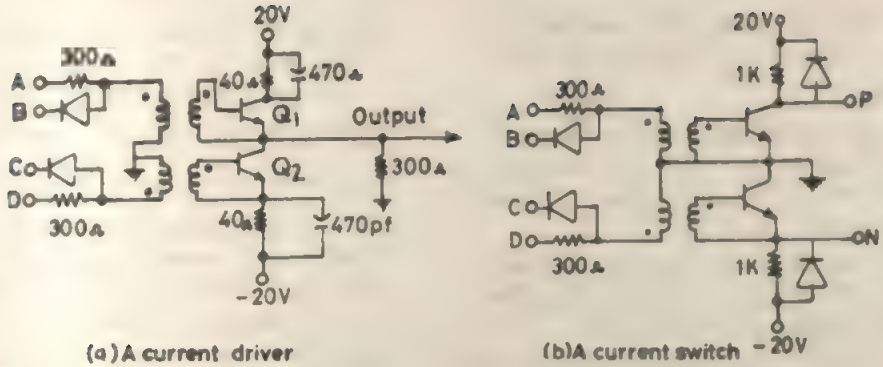



FIG.9-12 DRIVER/ SWITCH CIRCUITS

When the driver drives positive current (positive output voltage at the driver output), the current flows to the ground through the diode $D_2$ (of the selected wire) and the selected switch (which finally sinks it). The diode $D_1$ of the selected wire prevents this current from reaching the other wires. In case of negative current the roles of diodes $D_1$ and $D_2$ are reversed. The flow of current for selection of wire 14 is marked in Fig. 9.11. Typical circuits for drivers and switches are shown in Fig. 9.12 (a) and Fig. 9.12 (b) respectively.

To drive the positive current, a '0' (TTL level) is applied to the terminal A Fig. 9.12 (a) while keeping the terminal B at 1 level. This causes the current flow in the primary winding of the transformer. This in effect produces a positive base drive for transistor $Q_1$. The the transistor $Q_1$ turns ON and gives out the positive voltage at the output. The transistor $Q_2$ is kept OFF by 0's on the points C and D. The negative current is sent by interchanging the roles of terminal pairs A, B and C, D. In the similar way the switch circuit works *i.e.*, to sink the current of a given polarity, the appropriate transistor in the switch circuit is turned ON.

Using the selection technique discussed here, the number of drivers/switches required are considerably reduced. For a 3D coincident—concident memory of N locations the number of drivers/switches required for X and Y selection are 4. $N^{1/3}$ instead of $2N^{1/3}$ required in 3D memory of Fig 9.7. (Note that the switch and driver has the same circuit complexity).

## EXERCISES

1. How many cores shall be required to form a 4K × 8 memory.
2. How many cores shall be there in a plane of a 4K × 8, 3D memory.
3. How many bits are required to address 16K locations.
4. For a 4K × 8, $2\frac{1}{2}$D memory, how many rows and columns shall be there to form a core plane.
5. Find the bits to be allocated to X and Y parts of MAR for the memory in (4).
6. In a 64×4, 3D core memory, the $X_5$ wire (X select combination 101) of plane 2 (plane numbers 3 2 1 0) is broken.
   (*a*) Determine and explain the effect of this fault on read and write cycles by taking sample patterns.

(*b*) How many locations and bits (in the locations) are affected due to the fault.

(*c*) Assuming that the memory stored a pattern of 1111 in all the locations, find the data read from various locations.

(*d*) Execute the read cycle with 101010 in MAR. Put 0111 in MBR and execute the write cycle. Now execute the read cycle and give the contents of MBR.

7. You have a 64×8, 3D core memory. For each of the following failures (one at a time) in the system, give the complete behaviour of the memory, *i.e*, locations affected, bit affected, what is the effect, etc.

   (*a*) X wire number 4 has been broken

   (*b*) Y wire number 2 has been broken

   (*c*) driver number 6 cannot sent −ve half select current instead it sends +ve half select current.

   (*d*) MAR bit number 1 is stuck at 0

   (*e*) MBR bit number 3 is stuck at 0

   (*f*) Inhibit driver number 3 is faulty and it never sends any current.

8. For each fault given in (7) find out what happens when the pattern 11110000 is recorded in all the locations. (by executing read and write cycles).

9. Repeat (8) with the pattern 00001111.

10. Can you suggest a memory organisation for the core memory, where drive wires will be sending $\pm$Im/3 current only. Assume that cores remain by 2/3 Im.

11. Repeat (7) for a 256×8 2½D memory.

□

# 10

# SEMICONDUCTOR MEMORIES

## 10.1. INTRODUCTION

Since the early days of digital computers, ferrite cores have played an important role in the computer memory systems. However, with the advent of Large Scale Integrated (LSI) semiconductor technology, there is a significant impact on performance, cost, reliability and size of digital systems which use LSI technology, memory systems being no exception to this development.

The LSI technology is being increasingly used in complex digital systems like memories and processors. In this chapter, we shall discuss memory devices manufactured using LSI semiconductor technology. The use of typical commercially available memory chips shall also be illustrated in designing the larger memory systems.

## 10.2. TERMINOLOGY

In this section, we shall briefly discuss the important terminology and meanings of various frequently used terms in connection with the semiconductor memory devices.

### Dynamic Devices

These are the devices which need a periodic refresh, otherwise they lose the information. The term refresh means that we must put the information back in the device again and again at some specified minimum rate. The refreshing is required in dynamic devices, because the storage cells of these devices use capacitors for information (voltage) storage. Usually these devices are designed such that, if we access (read) the information from a device, it gets refreshed automatically. Thus in most cases, the refreshing involves reading the data at least once in a given period of time (the time, after which, the capacitors would have discharged below some minimum voltage). This time is usually a few milliseconds (typically 1 ms).

### Static Devices

Static devices do not require refresh and they can retain information as long as power is ON. The static memory devices use flip-flops as basic storage cells and therefore there is no need of refreshing. These devices loose the information when the power supply is put off (flip-flops shall not function because there is no power).

### Volatile Memory

A memory which loses its contents upon power OFF is called a volatile memory, while a memory which retains the information even when the power is turned off is called a non-volatile memory.

Semiconductor memories (except read only memories) are volatile, while core memories are non-volatile.

**Random Access Memories (RAMs)**

RAM is an array of storage cells organised such that any word (some fixed number of cells) can be accessed directly by supplying the address and executing read/write operation.

**Bipolar and MOS Devices**

Bipolar devices are the devices which operate with the help of currents contributed by both the types of carriers (electrons and holes).

The devices built using conventional transistors are bipolar, while those built using Metal Oxide Semiconductor (MOS) technology are not. The MOS devices have field effect transistors constructed using MOS technology.

## 10.3. BIPOLAR RANDOM ACCES MEMORIES

In this section, we shall discuss the basic storage cells and their organisations for bipolar RAMs. These cells are usually organised using either linear or coincident selection techniques.

### 10.3.1. Storage cells

Fig. 10. 1. shows the basic storage cells for bipolar RAMs The storage cell shown in Fig. 10.1. (*a*) is used in the linear selection organisation, while that shown in Fig. 10.1. (*b*) is used in the coincident selection organisation. In both the cases, a basic cell is made of DCTL (Direct Coupled Transistor Logic) flip-flop. The flip-flop is formed by transistors $Q_1$ and $Q_0$. The emitter ground connection required for flip-flop action is maintained by a logical 0 state of select line/lines

In Fig. 10.1. (*a*), X line constitutes the address line, which is usually (when the cell is not addressed) kept at a logical 0 state. This logical 0 provides a sink for current of an ON transistor. The cell flip-flop keeps the data till the address line goes to 1 state, at this time (when address line has 1 state), a new data could be written into the cell. The lines $B_1$, $B_0$ are read/write lines (bit lines) and their functions shall be discussed shortly.

In Fig. 10.1 (*b*), the cell has two select lines X and Y, both of these select lines must be switched to a 1 state to select the cell for a read/write operation. It can be very easily seen that by switching only one of the lines (X,Y) to logical 1, the other emitter would be getting the ground potential (0 potential) and therefore the cell will not be selected.

**Read operation**

To read the contents of the cell, all of its select lines are switched to a logical 1 state. This, thus, stops current flow of the ON transistor into the address line/lines. Now this current flows through one of the bit lines, $B_0$ or $B_1$ depending upon whether $Q_0$ or $Q_1$ is ON *i.e.*, whether a 0 or 1 is stored. The lines $B_0$ and $B_1$ are fed to sense circuit which shall be discussed shortly. The sense circuit gives out the data from the cell.

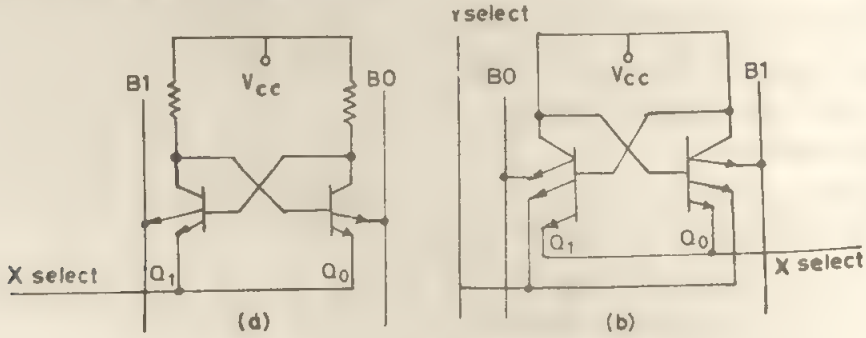



FIG.10-1 BIPOLAR RAM CELLS

**Write operation**

The write operation on the cell is carried out as follows. The cell is selected by making its select line/lines high (logical 1). If a 0 is to be recorded then lines $B_0$ and $B_1$ are put to logical 0 and 1 state respectively. This forces $Q_0$ to the ON state and $Q_1$ to the OFF state, thus recording a 0. Similarly a 1 is recorded by making $B_0$ and $B_1$ logical 1 and 0 respectively. After a small delay, the select lines are switched back to the 0 state and the cell is disabled, thus isolating it from further accesse.

### 10.3.2. Linear selection organisation

Linear selection bipolar RAMs are organised as a plane of n×k cells. Each row corresponds to one location of k bits. All the cells in a row are selected by a common row-enable signal which corresponds to the decoded address of the location. Fig. 10.2. shows 64×4 linear selection memory organisation with read/write electronics. The X select lines of cells in a row are connected together and are fed by the decoder output corresponding to the location, while $B_0$ lines of cells in a column are connected to a common $B_0$ line. Similarly $B_1$ lines are connected. Read circuit consists of sense amplifiers which are fed by $B_0$ and $B_1$ (bit lines). The output of the inverter fed by $B_1$ is taken as DO data out. The sense amplifier on $B_0$ is required to sink the current when a cell storing a 0 is selected. If this is not included, then the cell will lose the data because, no sink will be available to the current from the emitter of the ON transistor. The write circuit consists of two tri-state drivers ($TD_1$, $TD_2$) and their enable/disable logic. These tri-state drivers are enabled by the common line $R/\overline{W}$ (Read/Write). Drivers are enabled by a 0 single on $R/\overline{W}$ line, and

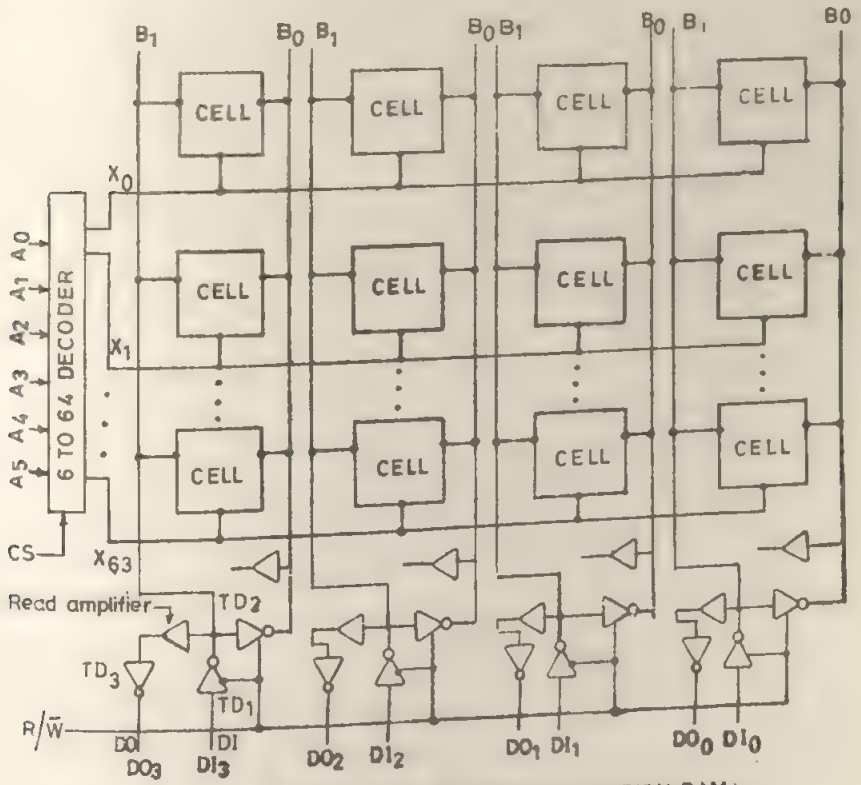



FIG. 10·2 64X4 BIPOLAR LINEAR SELECTION RAM

they pass the voltage from a DI (Data Input) line to the lines $B_1$ and $B_0$ in the inverted and true form respectively. The address decoder and Data out tri-state drivers ($TD_s$) are enabled by the Chip Select (CS) signal. If the chip select signal is not active the chip is not selected and data out lines remain in the high impedance state.

**Read operation**

To carry out the read operation, the R/$\overline{W}$ line is put in the read state (*i.e.*, logical 1), which disables the lines $B_0$ and $B_1$ from tri-state drivers $TD_1$ and $TD_2$. The emitter current flows through one of the lines $B_0$ and $B_1$ depending on whether a 0 or 1 is stored in the selected cell. This current is sinked by one of the sense amplifiers and the output of the selected cell is available on the line DO (Data Out). In Fig. 10.2, 4 cells (one row) are selected and their outputs are available on data out lines $DO_3$, $DO_2$, $DO_1$ and $DO_0$.

**Write operation**

To carry out the write operation, the addressed location is selected by the decoder (one of the X lines corresponding to the location becomes active) and after a small delay (few nanoseconds), the R/$\overline{W}$ line is

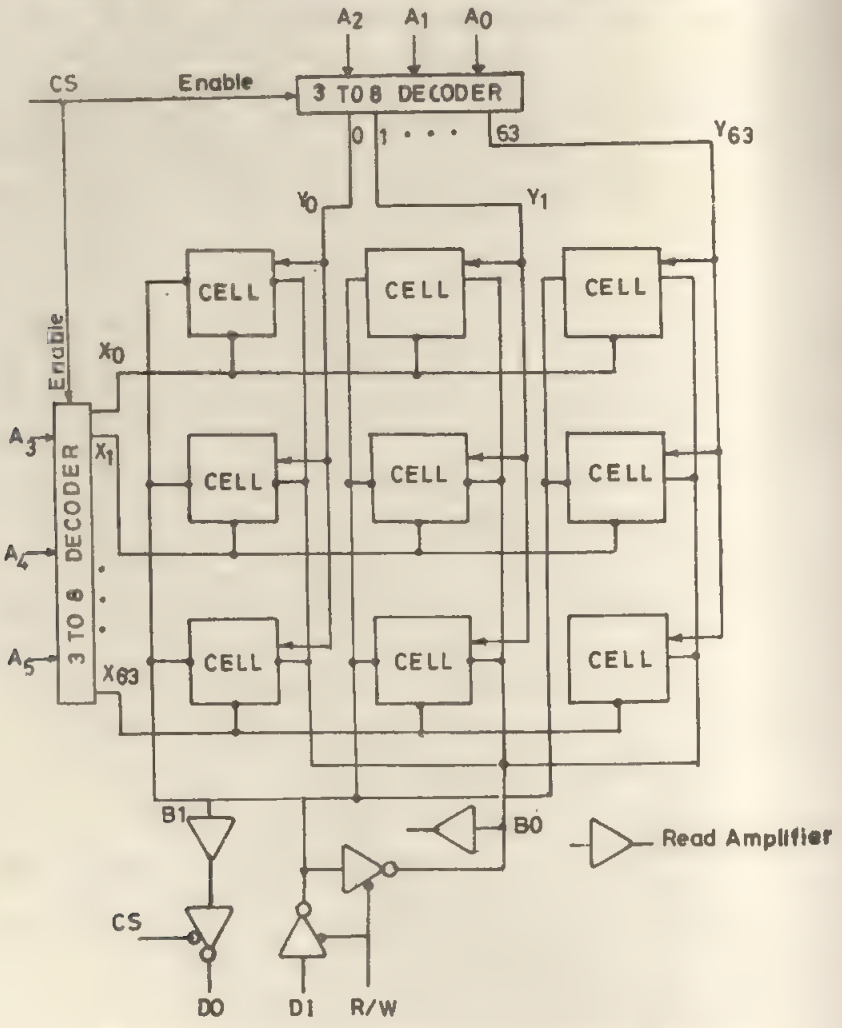



FIG. 10·3 64X1 BIPOLAR RAM WITH COINCIDENT SELECTION ORGANISATION.

made 0 (*i.e.*, active for the write operation). The data on the line DI is passed in the required form to the lines $B_1$ and $B_0$ through the tri-state drivers $TD_1$ and $TD_0$ as shown in Fig. 10.2. As per our earlier discussion, the data shall be recorded in the selected cells (addressed location). Note that when CS line is 1 (inactive) all the cells are disabled from being selected. This happens because, the decoder is not enabled since CS line is not active. Also in some memory chips, the CS lines may or may not control the output data drivers.

### 10.3.3. Coincident selection organisation

To build a large size memory using the linear selection organisation, large number of gates are required to build the address decoder ($2^k$ gates each with K inputs for a K—bit address). To avoid this, the memory cells are organised to form a coincident selection organisation. In this memory organisation, one plane of the cells constitutes one bit of all the locations. Fig. 10.3 shows a bit plane of 4k×1 coincident selection memory with storage cells of Fig. 10.1. (*b*). The input address lines are divided into two equal parts. One part addresses X select lines through the X decoder, while the other part addresses the Y select lines through the Y decoder. The cell at the intersection of the selected X and Y lines is selected. The other details are similar to that of the memory orgonisation discussed in Section 10.3.2.

## 10.4. MOS RANDOM ACCESS MEMORIES

MOS technology gives increased packaging densities and therefore more complex devices could be built using MOS technology, than that is possible with the bipolar technology, In this section, we study the basic MOS storage cells and their organisations as a part of larger memory systems.

In the MOS technology, there are two types of basic storage cells viz., (*i*) static and (*ii*) dynamic. We shall discuss both these in this section. Dynamic MOS memories always use coincident selection organisation because this gives better refreshing efficiency (accessing one cell in a row refreshes the entire row).

### 10.4.1. Static MOS RAMs

Fig. 10.4 shows the circuit diagram of a static MOS RAM cell used in the coincident selection organisation. This cell uses a flip-flop formed by cross coupled MOS inverter (P MOS). When the cell is not selected, the X

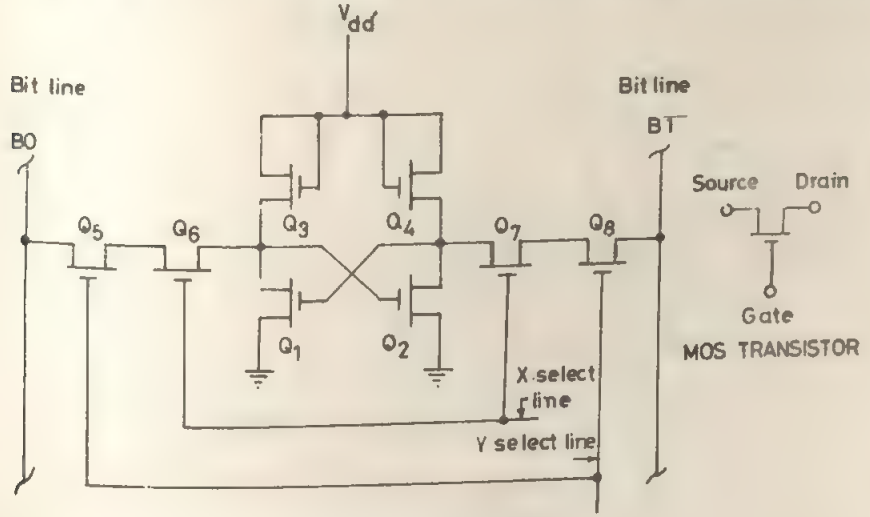



FIG.10·4 STATIC MOS RAM CELL

and Y lines are at ground potential. This potential on X and Y select-lines turns the transistors $Q_7$, $Q_8$, $Q_5$ and $Q_5$ off (a P MOS transistor is turned ON by a negative potential on the gate) The OFF states of these transistors keep the flip-flop formed by the transistors $Q_1$, $Q_2$, $Q_3$ and $Q_4$ ($Q_3$ and $Q_4$ act as loads for $Q_1$ and $Q_2$ respectively) isolated from the address lines as well as from the bit lines $B_0$ and $B_1$. This is true even if one of the address lines X and Y is at negative potential.

**Read Operation**

The cell is selected by giving a negative potential pulse to both the X and Y select lines. Suppose that the cell is storing a 1, *i.e.*, transistor $Q_2$ is ON and transistor $Q_1$ is OFF. Since the line X is negative, the transistor $Q_7$ will conduct and take the drain terminal of the transistor $Q_8$ to the potential on its source terminal (*i.e.*, ground since the transistor $Q_2$ is ON). Moreover since Y is also negative, it will cause transistor $Q_8$ to conduct and the line $B_1$ will experience a current because the source terminal of the transistor $Q_8$ is at ground potential. On the other side of the cell, transistor $Q_1$, $Q_5$ and $Q_6$ are OFF, and therefore the line $B_0$ will not experience a current. The current flow in the line $B_0$ or $B_1$ indicates the contents of the cell as 0 or 1 respectively. The currents in lines $B_0$ and $B_1$ are sensed by the read circuit and corresponding logical value (0 or 1) is presented to the output. Similar actions take place on the other side of the cell if it stored a 0.

**Write operation**

The write operation on a cell is carried out as follows. The cell is selected by giving the negative potential pulse on X and Y select lines. This gives a path for one of the write voltages on the bit lines to reach the drain terminal of one of the transistors $Q_1$ and $Q_2$. For example, if we want to record a 1 (*i.e.*, $Q_2$ ON and $Q_1$ OFF), the bit lines $B_1$ and $B_0$ are switched to ground and negative potential. Since the transistors $Q_7$ and $Q_8$ are ON (due to negative voltage on their gates), the ground potential of the line $B_1$ reaches the drain terminal of $Q_2$, and flip-flop is switched to 1 state irrespective of its earlier state, thus recording a 1. Similar actions take place on the other side of the cell when a 0 is to be recorded.

Fig. 10.5 shows a static memory cell used in the linear selection organisation. This cell is less complex (it has 2 transistors less) than the coincident selection memory cell. It has only one select line which is pulsed to the negative potential for the selection of the cell. The cell functions in exactly the same manner as that shown in Fig. 10.4.

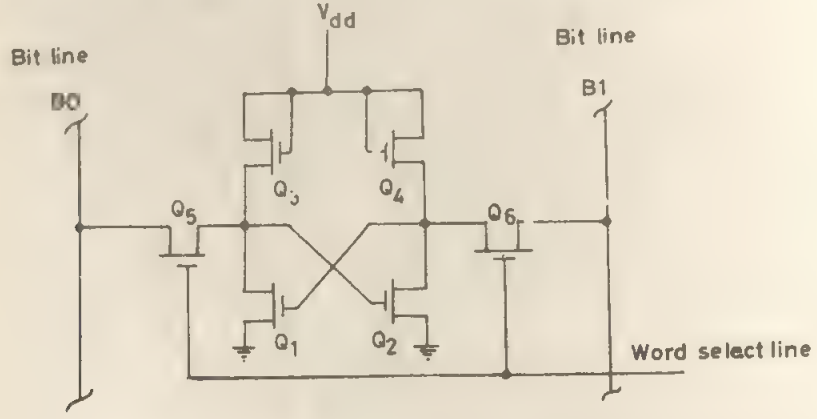



FIG. 10-5 STATIC MOS RAM CELL

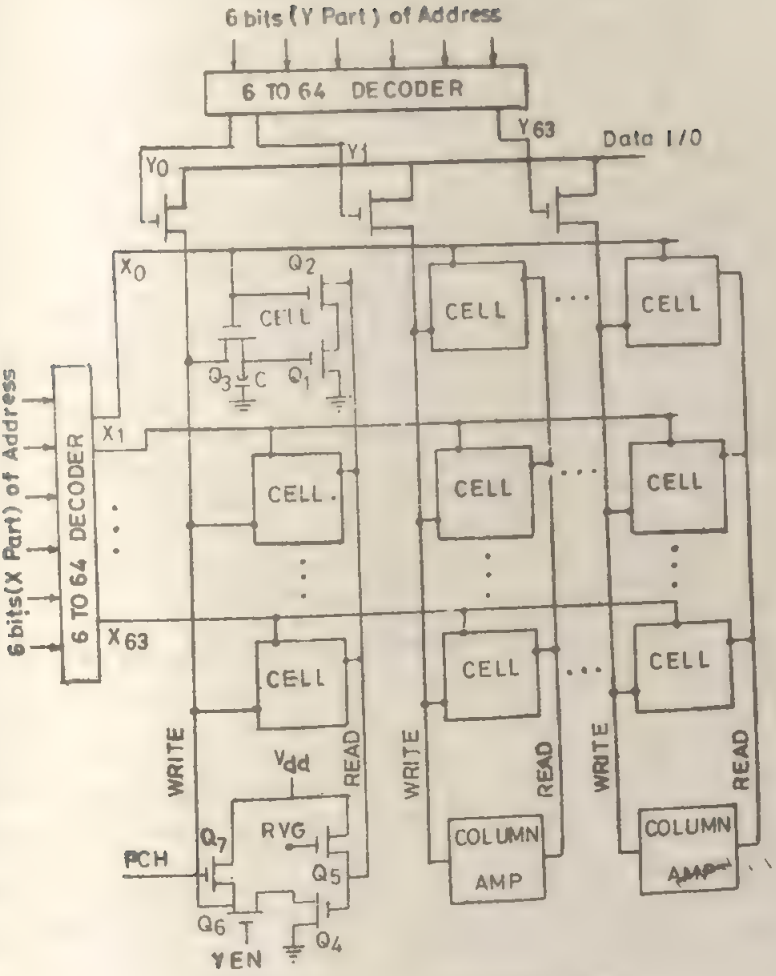



FIG. 10·6 4KX1 DYNAMIC MOS RAM

### 10.4.2. Dynamic RAMs

Dynamic MOS memory devices can be built with higher storage cells per mm² of the wafer area compared to the static MOS memory devices. This is possible due to the simplified structure of a dynamic memory cell. The storage element used in the dynamic memory cell is the parasitic capacitance. Fig. 10.6 shows a 4Kx1 dynamic MOS RAM (P MOS) with the coincident selection organisation. We shall briefly discuss this organisation to illustrate the basic principles involved in storage, access and refresh of the dynamic memories.

In Fig. 10.6, address lines $X_0$, $X_1$,... $X_{63}$ select one of the 64 rows of the cells. During an access to the memory, a full row takes part in the operation. It is one of the select lines $Y_0$, $Y_1$...$Y_{63}$ that finally selects the desired cell from the selected row, thus addressing a required cell uniquely. The X select signals are tri-state and vary from zero to mid-negative and most negative values.

The transistors $Q_1$, $Q_2$ and $Q_3$ forms the storage cell. Each column has 64 cells and at the bottom of the column, there is a column amplifier and other circuit composed of transistors $Q_4$, $Q_5$, $Q_6$ and $Q_7$. Normally

the transistors $Q_5$ and $Q_7$ are kept conducting by a negative potential (for P MOS) on the control lines PCH (precharge) and RVG. This keeps the READ and WRITE lines at a voltage slightly less than $V_{dd}$. During an access to the memory the potential on the PCH and RVG lines is removed. This makes the transistors $Q_5$ and $Q_7$ OFF, and thus READ and WRITE lines are disconnected from the $V_{dd}$ supply. At this time the selected X line is at mid-negative potential.

Assume that the capacitor C of the selected cell is storing a negative electric charge (storing a logical 1). This negative voltage is enough to turn ON the transistor $Q_1$. Also the negative voltage on the X select line turns ON the transistor $Q_3$. The transistor $Q_3$ cannot conduct because its gate potential is not negative enough due to the charge on the capacitance C. Since the transistors $Q_1$ and $Q_3$ are ON, the READ line discharges to the ground potential ($-V_{dd}$ connection to the READ and WRITE lines was already broken due to the OFF states of $Q_5$ and $Q_7$ respectively). On the other hand, assume that the capacitor C has no charge (storing a 0). This will keep $Q_1$ OFF and therefore READ line will not be discharged to the ground potential. As can be seen from the above discussion the state of the READ line at this time gives the complemented state of the data bit stored in the cell. Note the actions just discussed take place for all the columns. The state of READ line is transferred through the column amplifier to the WRITE line as follows.

When the cell stores a 0 (*i.e.*, READ line is not discharged), the transistors $Q_4$ and $Q_6$ conduct due to negative potentials on READ and YEN (YEN is made negative) lines. This will discharge the WRITE line to the ground potential through the conducting transistors $Q_4$ and $Q_6$. On the other hand, if the READ line is at ground potential (a 1 in the cell), the transistors $Q_4$ cannot conduct and WRITE line remains charged. It can be thus observed that the WRITE line is at complementary state to that of the READ line. The state of one of the sixty four WRITE lines is read on to the DATA line by the enable signal on the required Y select line. At this time the selected X line goes back to the third state, providing more negative voltage on the gate terminal of the transistor $Q_3$ and thus allows it to conduct irrespective of the voltage on the storage capacitor. This causes the transfor of voltage of WRITE line to the capacitor line, thus refreshing the cell automatically.

To put the external data bit into the required cell, the potential on the DATA line is transferred to the required WRITE line through the column transistor, made ON by the active state (high potential) of the selected Y line. The data on the selected WRITE line is transferred to the storage capacitor of the selected cell during the third high state (most negative state) of the X select line as discussed earlier.

**Refreshing of Dynamic Memories**

From the previous discussion it is to be noted that all the cells in the selected row take part in the read operation and their contents are read on READ and WRITE lines in the complemented and true forms respectively Further, the data on WRITE lines is written afresh into the selected cells (selected row of cells) and therefore all these cells are refreshed. Thus whenever a read operation is performed on any cell in a row, the cells in the entire row are refreshed. Therefore refresh cycles of the dynamic memories thus involve selection of X lines one after another for read operation. A refresher circuit therefore could be built which carries out the read cycles on rows at some specified rate. Whenever a refresh cycle is being carried out by the refresher, memory should not be allowed the access by any other device.

A 4k x 1 dynamic RAM arranged as a 64 x 64 cells would require 64 read cycles to refresh the entire 4k cells. Assuming a cycle time of 500 nanoseconds (which is typical of these memories) and the refresh rate of 1 millisecond (typical), the total time for which the

memory is unaccessible due to refresh is 64 x 500 nanoseconds, *i.e.*, 32 microseconds. This works out to be 3·2%. The percentage for the currently available largest dynamic RAM (64k x 1) is only 12·8% These percentages being negligible, the dynamic RAMs provide the most economical large size main memories for the digital computers.

## 10.5. READ ONLY MEMORIES

Read Only Memories (ROMs) store data of a permanent nature and we can operate these memories in the read mode only. ROMs are used in a number of applications in digital systems. In CRT displays ROMs are used to store the dot patterns of alphanumeric characters. In digital computers ROMs are used to store permanent data or programme. Also a number of useful combinational logic functions can be implemented by storing their truth tables in ROMs. Here, a location of ROM storing a value of the switching function is accessed using the combination of input variables of the switching function as the address.

ROMs are available in both bipolar as well as MOS families. The MOS ROMs are available as high capacity, low cost and slow speed devices. On the other hand, the bipolar ROMs are available as comparatively small capacity, high cost but high speed devices. The typical speed (access time) ratio between bipolar and MOS ROMs is ten.

The ROMs are fabricated such that the required data is written (programmed) when they are manufactured. The field programmable ROMs (PROMs) are also commercially available and they can be programmed by users as per their requirements.

### 10.5.1. Bipolar Read Only Memories

Typical bipolar ROM cell is shown in Fig. 10.7. It consists of a transistor whose base is driven by the select line. The emitter of the transistor is connected to the bit line only if the cell stores a 1, otherwise it is not connected. When the select line is made active (high), the transistor conducts and gives out a high (logical 1) voltage on the emitter. If the emitter is connected to the bit line, its voltage reaches the bit line giving out the high potential (logical 1) on the bit line.

These ROM cells could be organised as shown in Fig. 10.8 to form a 16 x 4 bit ROM. In this ROM, each row has four transistors (for 4-bit word). The base terminals of these transistors are driven by a common word select line. The row of the cells is enabled by a high potential on the selected word line (other word lines have low potential) and therefore the emitter base junctions of the transistor in the selected row are forward biased, thus causing the data of the selected location to appear on the bit lines. The table 10.1 gives the data stored by the ROM of Fig. 10.8.

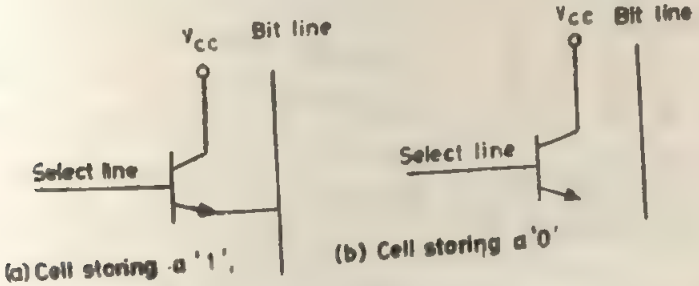
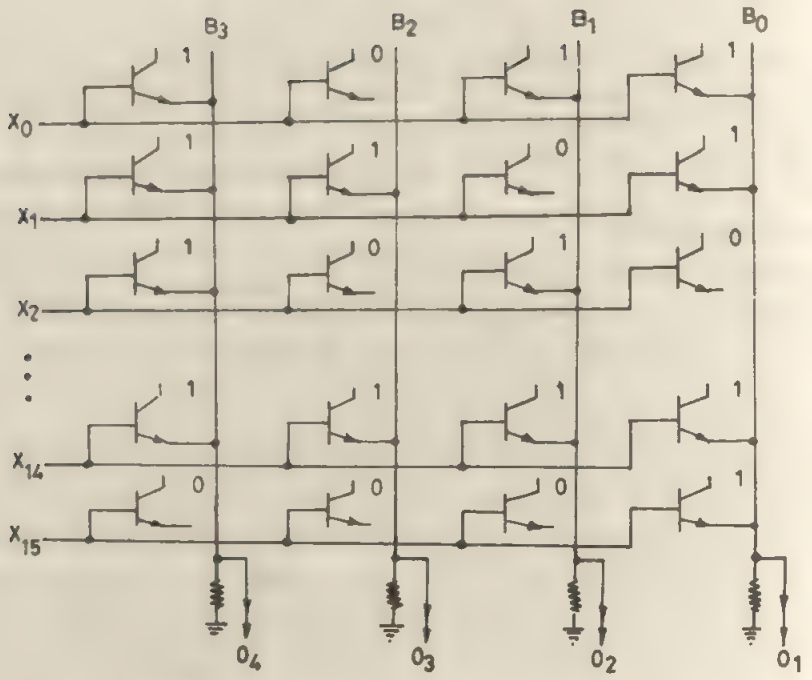



FIG. 10.7 BIPOLAR ROM CELLS




NOTE :- Collectors of all transistors connected to $V_{cc}$

FIG.10-8 16X4 BIPOLAR ROM

**TABLE 10.1. : ROM Data**

| Location | Bits | | | |
|---|---|---|---|---|
| | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 |
| ⋮ | | | | |
| 14 | 1 | 1 | 1 | 1 |
| 15 | 0 | 0 | 0 | 1 |

The lines $X_0$, $X_1$ ... $X_{15}$ are driven by the 4 to 16 decoder (not shown) which decodes the 4-bit address.

From the Fig. 10.8, it can be noted that all the four transistors in a row have their collectors connected to a common supply line $V_{cc}$. Similarly all the base terminals are connected to the common X select line. Because of this, the functions of all the transistors in a row can be carried out by a single multi-emitter transistor. The number of emitters thus shall be equal to the number of bits in the word. Fig. 10.9. (*a*) shows a 4-emitter transistor carrying out the functions of the row $X_1$ of transistors of Fig. 10.8. In programmable ROMs (PROMs), the emitter to bit line connection is made of a nickel chromium fuse which can be burnt by applying the programming pulse to the bit line after selecting the required word. To illustrate the PROM burning (programming) principles, consider the circuit shown in Fig. 10.9. In this circuit when the transistor is selected, its base has a logic 1 (high) voltage. A programming pulse of 8 volts is applied to the required bit line (the bit of the selected location where the fuse has to be blown). This voltage pulse causes the reverse

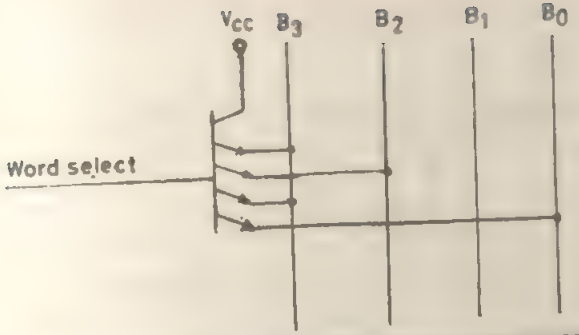



FIG. 10·9 (a) A MULTI-EMITTER TRANSISTOR STORING '1101' PATTERN

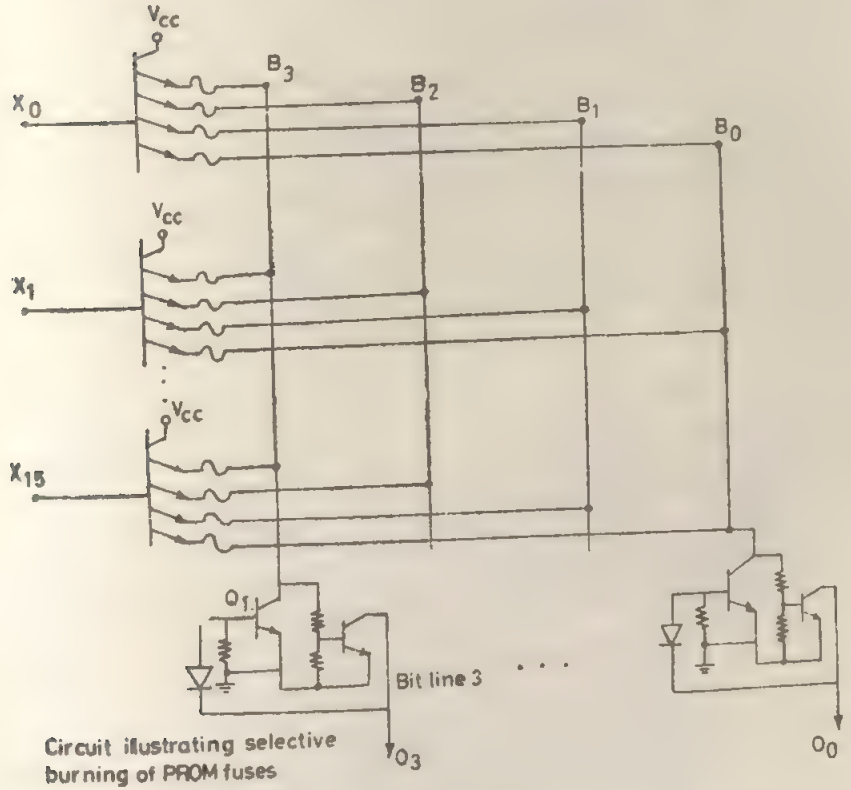



FIG. 10·9 BIPOLAR PROM INTERNAL CIRCUIT

breaks down of the diode shown in the circuit and the transistor $Q_1$ turns ON, thus effectively grounding one side of the fuse. The $V_{cc}$ of the entire circuit is raised to 12 volts at the same time. If the cell is selected (select line high), the fuse will carry sufficient current and get burnt.

Suppose bipolar PROM initially stores all 0 s. To program a 1 in the required bits, the programming procedure specified by the manufacturer has to be followd. The circuit actions discussed here about the PROM programming are typical and there could be variations depending on the manufacturers.

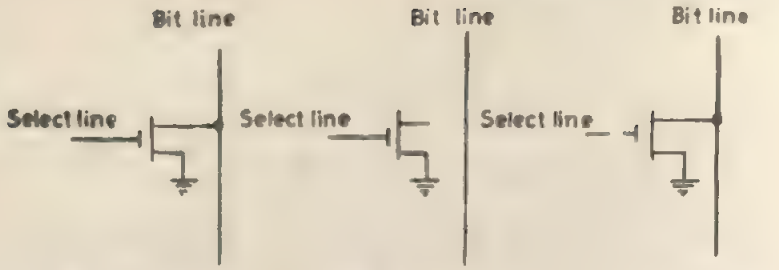



FIG.10·10 MOS ROM CELLS

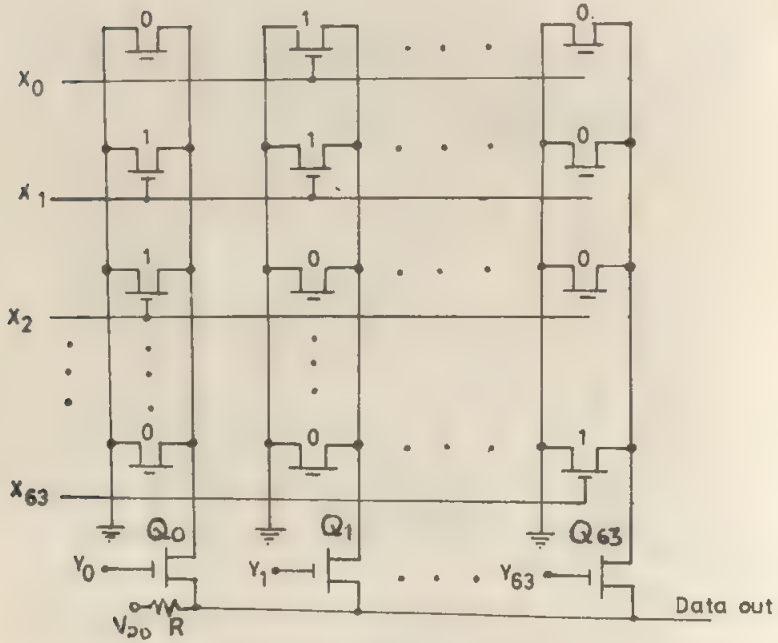



FIG. 10·11 4K X 1 MOS ROM

### 10.5.2. MOS read only memories

Typical MOS ROM cells are shown in Fig. 10.10. In Fig. 10.10. (*a*) the cell storing a 1 has its gate connected to the select line, while in Fig. 10.10. (*b*), the gate is not connected and thus the cell stores a 0. Other arrangement to store a 0 is shown in Fig. 10.10 (*c*). During the read operation the cell is selected which causes the bit line to experience a current or no current depending on the contents of the selected cell.

These cells can be organised as shown in Fig. 10.11. to form a coincident selection memory. The bit line of the selected column carries a high potential due to the ON state of one of the transistors $Q_0$, $Q_1$ ... $Q_{63}$. Since this potential is available through the resistance R, any ON transistor (only selected transistor can be ON) in the column can reduce the bit line potential to ground and consequently data line goes to the ground potential, and the data from the selected cell is read out.

It may be noted that only the selected cell transistor in the column if ON can reduce the bit line potential to ground potential and consequently data line goes to the ground potential. If the selected transistor is OFF, the data line carries the high potential. In this way the cell selection and data read operations are performed.

## 10.6. THE EXAMPLES OF MEMORY CHIPS

In this section, a few commercially available semiconductor memory devices and their usage in the design of memory systems are discussed.

### 10.6.1. The 8111 static MOS ram

The 8111 is a 1024-bit static MOS random

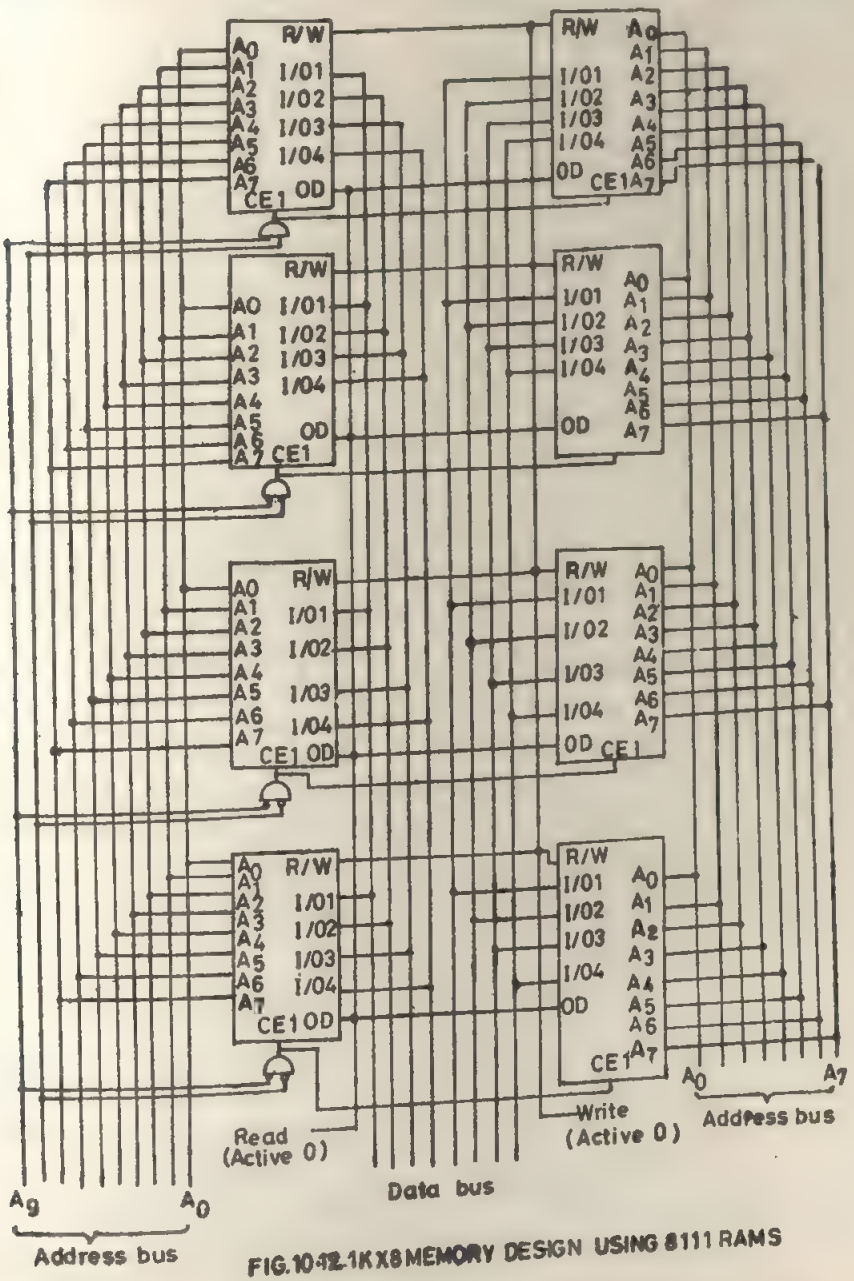



FIG. 10.42. 1K X 8 MEMORY DESIGN USING 8111 RAMS

access memory fabricated using N channel MOS technology. It is organised in 256 words each word having 4 bits. It has a maximum access time of 450 nanoseconds. All of its input-output pins are TTL compatible and it operates with only one +5 volt power supply. The pin diagram and the logic of the chip could be understood from Fig. 10.12, which shows the use of these chips.

**Functional Pin Description**

The pins marked $A_0$, $A_1$ ... $A_7$ provide the address of a 4-bit word to be accessed. The $CE_1$ and $CE_2$ are chip select lines and both of them must be made active (*i.e.*, logical 0) to select the chip for read or write operation. The R/W (READ/WRITE) line is normally in the logical 1 state indicating the read operation. It is made 0 only when the data is to be recorded in the memory. The data from the memory is available on the I/O pins $I/O_1$, $I/O_2$, $I/O_3$ and $I/O_4$ only if the line OD is made 0. Otherwise (OD=1) even if the chip is selected and the R/W line is 1, the I/O pins do not carry the data, instead they remain in the high impedance state. The high impedance states of I/O pins make it possible to tie these pins to the similar pins of other devices.

**1 K×8 Memory Design Using 8111 Chips**

Fig. 10.12 shows a design of 1 K×8 memory system using these chips. In Fig. 10.12, we use 2 chips in a row to form 256×8 words of memory. Four such rows forms memory of a 1 K×8 size. The 1 K×8 memory requires a 10 bit address. The eight least significant address lines ($A_0$, $A_1$ ... $A_7$) of the 10-bit address are connected to the eight address lines of all the 8111 chips. While the two most significant address lines $A_9$ and $A_8$ are connected to the chip select terminals of 4 rows in true/complemented form as shown in Fig. 10.12. These two lines decides the selection of a row. The external READ and WRITE control lines signal the memory for read and write operations respectively. READ line is connected to OD pins of chips while WRITE line is connected to R/W pins of chips. The address and data lines usually are connected to the system address and data bases. Chip enable $CE_3$ (not shown) can be used for extension.

### 10.6.2. The 2114 Static RAM

The 2114 is a 1024×4-bit static MOS random access memory fabricated using N-channel MOS technology. It is available in various models with maximum access times varying from 200 to 450 nanoseconds. It has all pins TTL. compatible and have common data input and output lines. These lines have tri-state capability and thus could be used easily in the bus configurations where data lines of a number devices could be tied together. The logic symbol with pin configuration of the 2114 IC package is shown in Fig. 10.13.

**Functional Pin Description**

The lines $A_0$, $A_1$ ... $A_9$ provides the 10-bit address of a 4-bit word to be accessed. The CS line when 1 disables the chip being accessed. The line WE (Write Enable) when 0 indicates the write operation. The CS and WE lines are normally (when no access is required to the memory chip) at 1 state. Data is recorded in the memory by establishing first the desired 10-bit address on the address lines After this the WE line is made active *i.e.*, 0 and then CS line is made active (logic 0). All the lines (including the address lines) are kept at fixed level for some time (specified by the manufacturer). This will cause the data on the I/O lines to be recorded in the desired location. After this time, the CS and WR lines should be made high. To read a location, the address is established on the address lines and the CS line is made 0 (WE should be at 1).

The use of these chip in the larger system is left to the reader as an exercise.

### 10.6 3. The 2116 Dynamic RAM

The 2116 is 16 K×1 dynamic RAM providing a very inexpensive memory for digital systems. It is available in a 16 pin package with all the pins TTL compatible. It requires

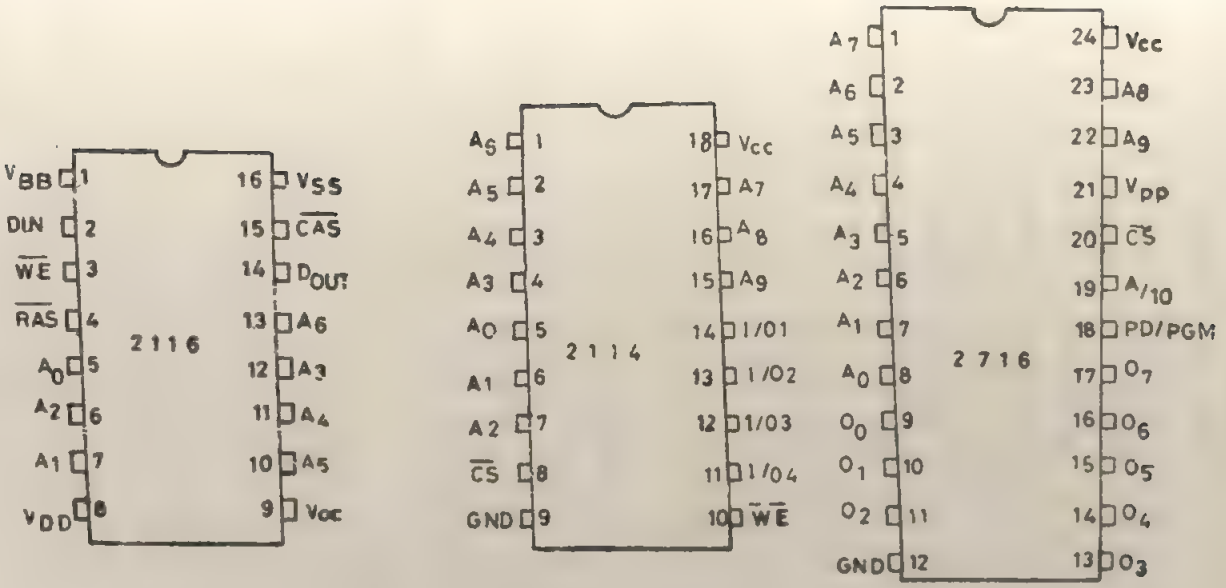



FIG. 10·13 PIN DIAGRAMS OF RAM/ROM CHIPS

three power supplies +12 V, +5 V and −5 V for its operation. Since it is a dynamic RAM, it requires refreshing every 2 milliseconds. The functional pin diagram and logic symbol are shown in Fig 10.13.

The 2116 RAM has only 7 address lines. The external 14-bit address is to be given to 2116 in two steps through a 2 to 1 multiplexor (7 line in each group). The RAM is arranged as an array of 128 rows and 128 columns of dynamic storage cells. The RAS signal is active low and latches the 7-bit address into the ROW register of device. Similarly the CAS (active low) signal latches the other 7-bits of address given on the address lines $A_0$, $A_1$, ... $A_6$ in the column register of the device. These two registers in the device together provide a 14-bit address. The WE (Write Enable) signal when active (logical 0) executes a write cycle for the memory and data is recorded in the addressed location of the memory.

Since the memory uses dynamic storage cells, it needs periodic refresh at every 2 milliseconds. The refreshing can be achieved by read cycles carried on the rows addressed by $A_0$ through $A_6$. Other refreshing modes are also possible and may be found in the product data sheets.

### 10.6.4. The 2716 EPROM

The 2716 is ultra-violet erasable and electrically programmable read only memory. It operates with a single 5 V supply. It has a 16K-bit storage cells arranged as 2K 8-bit words. It has a maximum access time of 450 nanoseconds. The functional pin diagram and logic symbol of this memory device are shown in Fig. 10.13. The lines $A_0$, $A_1$...$A_{10}$ are the 11 address lines required to address one of the 2048 locations. The data from the addressed location is available on the 8 data-out lines $O_0$, $O_1$,...$O_7$. There is single chip select signal to select the chip for operation. The pins marked $V_{pp}$ and PD/PGM are used to program (burn) the memory. When the chip is not selected ($\overline{CS}=1$), the output lines are tri-stated (high impedance) and thus, this device could be used easily for the data transfer through a common bus which can be used by other devices.

The line PD/PGM when at logical 1 causes the chip to be in the power down mode. In this mode the chip takes only 25% of the normal power (normal power is 525 mW) requirements.

#### Programming of the 2716 Eprom

Initially and after each erasure (*i.e.*, exposer to the ultra-violet light) all bits of the PROM go to the 1 state. The required data is introduced by selectively programming 0's into the desired locations. The chip can be put into the programming mode by connecting 25 Volts to the pin $V_{pp}$ and applying a logical 1 voltage to the CS line. The data to be programmed is applied 8 bits in parallel to the data-output pins $O_0$, $O_1$...$O_7$. The levels required for data and address inputs are at TTL level. When the address and data lines are stable, a 50 millisecond, active high (TTL) programming pulse is applied to the PD/PGM pin. In this way one location gets programmed. To program another location, we put the required address and data on the corresponding pins and apply again the programming pulse. The locations can be programmed in any order. After the programming is complete, the $V_{pp}$ line must be connected back to the + 5 V supply. Note that if a PROM is already containing a 0 in some bit, then a 1 cannot be programmed in that bit by the programming procedure just discussed. To carry out this the entire memory has to be erased by sufficient exposer to the ultra-violet light, and then re-programmed.

## EXERCISES

1. Is it possible to arrange the bipolar ROM cells (Fig. 10.9) to form a 64 x 4 bits coincident selection ROM. If yes, then work out the design schematic
2. Using 8111 chips work out the design of 4K x 8 random access memory.
3. Repeat (2) for 1 K x 16.
4. Using the 2116 dynamic RAM chips discussed, design a 64K x 8 RAM.
5. From any book (from the list of references given) dealing with MOS transistor circuits, study the working and the use of MOS transistors in digital circuits.
6. Design the refresher logic for the memory designed in (4). Give the complete logic schematic of the memory along with the refresher.
7. Can you think of a simpler dynamic RAM cell (cell with fewer transistors than that used in the cell of Fig. 10.6). If yes, work out the circuit details of the cell and show how these cells could be used in the linear and coincident selection organisations forming larger memory systems.
8. Using the linear selection bipolar RAM cells of Fig. 10.1 work out the organisation of these cells to form a 64 x 1 coincident selection RAM.
9. In a coincident selection bipolar RAM of 64 x 4 capacity, the X decoder number 3 (X line number 3) is faulty and always remains at 0. Analyse and bring out the effects of this fault on read/write operations.
10. A 64 x 4 bipolar RAM has a stuck at 0 fault on the some address line (MAR bit). By reading and writing various data patterns (you have to choose them), indicate how you will detect and locate this fault.
11. Repeat (10) for the faults on :
    (a) decoders
    (b) memory cells
12. Compare core, MOS static, MOS dynamic and Bipolar RAMs on any four counts.

□

11

# INTRODUCTION TO DIGITAL COMPUTER ARCHITECTURE

## 11.1. INTRODUCTION

A digital computer executes a program composed of a series of machine language instructions resident in the main memory. These instructions and data are coded as binary numbers. The machine language and its execution speed, and various other facilities available within the machine, determine the computing power of a CPU and the computer. The architecture of a machine is its appearance to the machine (assembly) language programmer, *i.e.*, what kind of facilities can be coded in the machine language, how many registers are available, how data is processed and what facilities are available to carry out the data transfer between I/O devices and the main memory.

## 11.2. CPU AND ITS FUNCTIONS

A CPU executes the machine language program composed of a sequence of instructions. It executes instructions one by one. To understand how a CPU executes an instruction, consider the simplest form of the instruction coding shown as follows :

| Field 1 | Field 2 |
|---|---|
| Operation Code (Opcode) | Operand Address |

An instruction in the above format, has two parts (fields). In the first field, the operation to be performed is coded (by a number), while the second field gives the address (another number) of the main memory location, where the operand taking part in the operation resides. The CPU performs the following actions to execute an instruction :

(*a*) It reads the instruction from a main memory location (indicated by some pointer) into the CPU, and increments the (instruction address) pointer.

(*b*) It decodes the instruction *i.e.*, finds out what is to be done.

(*c*) It gets the operands if required (this will depend on the opcode) from main memory.

(*d*) It performs the operation indicated by the operation code.

To carry out the above steps, one could easily think of the components CPU should have. We list them below :

(*a*) A register to hold the address of the next instruction. This register is called Program Computer (PC) or Instruction Address Register (IAR). This register is incremented every time an instruction is fetched.

(*b*) One register to hold the current instruction. This register is called the Instruction Register (IR).

(*c*) One or more registers to hold the operands and the result of the operation. A register which is used to hold an operand and also the result of the operation is called the Accumulator. (AC) Many instructions implicitely treat the contents of this register as one of the operands.

(*d*) An instruction decoder to decode the operation code of an instruction present in the IR. A functional unit *i.e.*, the arithmetic logic unit (ALU) to carry out the operation.

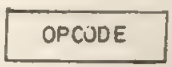
(*f*) A sequencer to carry out the proper sequencing of steps required to execute an instruction.

## 11.3. INSTRUCTION FORMATS

Each machine instruction specifies a particular operation to be performed by the CPU. The simplest coding of an instruction may be as discussed in Section 11.2. This format is called single address format. Other instruction formats are also possible and some of them are shown in Fig. 11.1.

Some machines may have the instructions with fixed length (*i.e.*, number of bits) while others may have variable length instructions. Moreover instructions may be of shorter, same or longer lengths than the word length of a machine. The number of bits in an instruction format is usually chosen as an integral multiple of the smallest addressible unit of the information in the main memory.

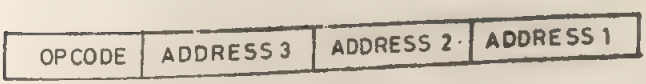
Consider a machine having $s+k$ bits long single address instructions, where $s$=number of bits in the opcode field and $k$=number of bits in the operand address field. Such a format allows $2^s$ different operation codes and $2^k$ locations of the addressible memory. Alternatively an $s+k$ bit instruction word could be broken into $s-1$ bit opcode and $k+1$ bit operand address field. This will give half




(a) 0-address format


(b) 1-Address format


(c) 2-Address format




(d) 3-Address format

FIG 11.1 INSTRUCTION FORMATS

the number of instructions but twice as much memory addressible. Some computers use a different kind of trade-off between the address and opcode bits. This trade-off leads to the new scheme called expanding opcode instruction formats which is discussed in the following paragraphs. The concept of an expanding opcode format could be best explained through an example.

Consider a machine which has a 12-bit, 3-address instruction format with a 3-bit operation code and 3 bits in each of the address fields. One design would be to have 8 instructions, each with 3-addresses. Suppose we want only 7 instructions with a 3-address format, then one opcode combination, say 111, could be used to have more instructions with 2-addresses (say address 1 and address 2). The bits of an unused address field could be used along with the 111 combination of the opcode bits to form a larger size opcode. In this way we shall have 8 additional instructions with two addresses. Again, if the designer wants only say 6 double address instructions then, the two unused combinations of bits 8, 7 and 6 (Table 11.1) could be used along with address 2 field bits to form 16 instructions with single address. Further if only 15 single address instructions are required, one can code 8 more instructions with zero address.

TABLE 11.1

**Expanding Opcode Format**

| 11 10 9 | 8 7 6 | 5 4 3 | 2 1 0 |
|---|---|---|---|
| Opcode | Address 3 | Address 2 | Address 1 |
| 3-bit Opcode | Address 3 | Address 2 | Address 1 |
| 000<br>001<br>010<br>011<br>100<br>101<br>110 | 7 instructions with 3—addresses | | |
| 111 | 000 | 6 instructions with 2 addresses | |
| 111 | 001 | | |
| 111 | 010 | | |
| 111 | 011 | | |
| 111 | 100 | | |
| 111 | 101 | | |
| 111 | 110 | 000 | 15 instructions with one address |
| 111 | 110 | 001 | |
| 111 | 110 | 010 | |
| 111 | 110 | 011 | |
| 111 | 110 | 100 | |
| 111 | 110 | 101 | |
| 111 | 110 | 110 | |
| 111 | 110 | 111 | |
| 111 | 111 | 000 | |
| 111 | 111 | 001 | |
| 111 | 111 | 010 | |
| 111 | 111 | 011 | |
| 111 | 111 | 100 | |
| 111 | 111 | 101 | |
| 111 | 111 | 110 | |

| 12-bit Opcode | | | |
|---|---|---|---|
| 111 | 111 | 111 | 000 |
| 111 | 111 | 111 | 001 |
| 111 | 111 | 111 | 010 |
| 111 | 111 | 111 | 011 |
| 111 | 111 | 111 | 100 |
| 111 | 111 | 111 | 101 |
| 111 | 111 | 111 | 110 |
| 111 | 111 | 111 | 111 |

8 instructions with no address field.

This thus gives us a total of 7+6+15+8=36 instructions.

The scheme discussed is summarised in Table 11.1.

The expanding opcode formats are popular and are used in a number of machines small and big. Popular computers which use expanding opcodes are PDP 11 and HP 2100 series of minicomputers manufactured by Digital Equipment Corporation (DEC) and Hewlett Packard respectively, and Cyber 70, a large scale scientific machine manufactured by Control Data Corporation (CDC).

## 11.4. ADDRESSING MODES

An address field in the instruction format tells the machine about the operand. The bits in the operand field could be used in a number of ways, giving rise to the various addressing modes. The design of addressing mechanisms constitutes the major architectural design of computers. In this section, we shall discuss various addressing techniques which are commonly used in various computers.

### 11.4.1. Direct Addressing

This is the simplest and conventional addressing technique. It is available in some form or other on almost all the computers. In the direct addressing, a number contained in the address field gives directly an address of the memory location. For example, the following instruction refers to the operand in location 200.

| Opcode | 200 |
|---|---|

### 11.4.2. Indirect Addressing

In indirect addressing, the address field specifies the memory location which contains, not the operand but the address of the operand *i.e.*, the contents of the location specified by the address field are obtained and is treated as the address of the location and the data from this new location is accessed. In some machines, further indirection is available, *i.e.*, address obtained from the memory location may specify further indirection. (One bit, say most significant bit, may be coded as 1 for further indirection). We shall illustrate indirect addressing by the following examples.

Consider the instruction shown in Fig. 11.2 (a). This instruction refers to the location 4000 indirectly through the location 500. The data referred by this instruction is 6532 (contents of location 4000). Another instruction shown in Fig. 11 2 (b) refers to the location 3000 indirectly through locations 5000 and 200. In this case the contents of the location 5000 specifies one more level of indirection, thus giving 3000 as an address. The memory references and final data accessed by these instructions are shown in Fig. 11.2. (D : Direct, I : Indirect).

Indirect addressing is useful in many ways. For example, we could use indirect addressing as follows to access the data in a table sequentially by having one level of indirect addressing.

A pointer to the starting address of the table is kept in some location say X. The

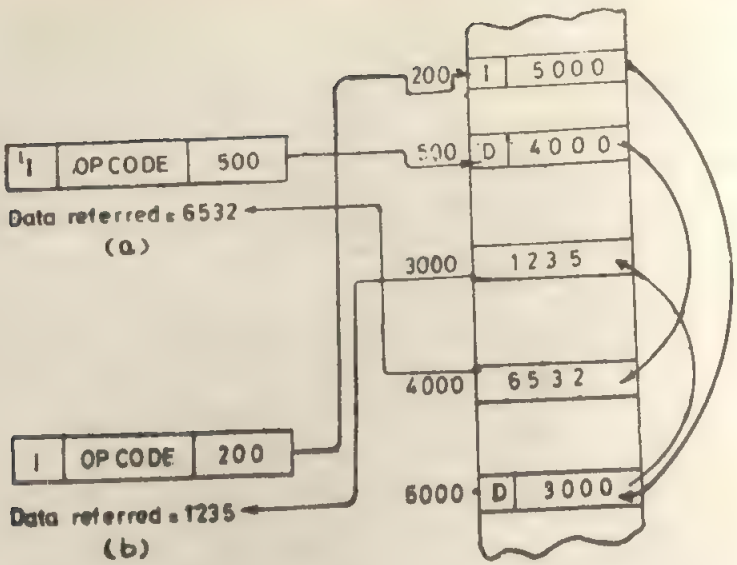



FIG 11·2 ILLUSTRATION OF INDIRECT ADDRESSING

elements of the table are accessed by referring to the location X indirectly (X is incremented after every reference) as illustrated in the following program schematic.

```
      ⋮
500   ADD 200, I
      ⋮
      INC 200
      ⋮
      JMP 500
```

Another use of indirect addressing is that we can address larger memory with a small number of bits in the operand address field. For example, consider a machine with 16-bit word length, and instruction word of 16-bit with 10 bits for operand address field. Using direct addressing we can address 1K (1K= 1024) memory locations. If the same operand address field has 1 bit for indicating direct/ indirect addressing, and 9 bits for direct address, we can address 512 locations directly, but any 512 locations from $2^{16}$=64K locations indirectly through the pointers (addresses) stored in the locations 0 to 511.

### 11.4.3. Register Addressing

The address field in this type of addressing specifies the address of a register which contains an operand. Register addressing is available on machines having a number of registers. Since registers are local to CPU and are made of very fast memory devices, the execution speeds of instructions operating on data in registers are high.

### 11.4.4. Register Indirect Addressing

Unlike the register addressing in register indirect addressing, the addressed register does not contain the data, instead it contains the address of the memory location holding the operand. Since there are few register (at most say 16), length of the address field is small but main memory addressing space is large depending upon the length of the registers. The Intel 8080/8085 based microcomputers have large number of instuctions which use register indirect addressing.

### 11.4.5. Index Addressing

In index addressing, the address field of an operand has two subfields. One subfield specifies one of the registers, while the other subfield specifies the displacement. These registers are called index registers. The

operand address is calculated by adding the contents of the specified index register to the number coded in the displacement field.

Index addressing is very useful and is found on many computers. It is useful in accessing tables and matrix elements. Also a larger memory could be addressed with an operand field having considerably smaller number of bits. Of course, this depends upon the length (number of bits) of registers.

### 11.4.6. Base Addressing

The computation of operand address in base addressing is exactly the same as that in index addressing. In base addressing, one operand address subfield specifies a register called base register, while the other subfield gives the displacement. Although the operand address calculation is same as that in the index addressing, the purpose for which base addressing is provided is different.

Base addressing is helpful in program relocation, *i.e.*, a program code can be moved in memory from one area to another and can be successfully executed. For example, consider a program schematic shown in Fig. 11.3. Let the contents of the register B be 1000. An instruction JUMP B, 500 transfers the program control to the location given by the sum of the number in B register and the displacement 500, *i.e.*, to the location 1500.

Now suppose, we moved the program with its starting point at the location 2000. When the program is executed in this new area, the JUMP B, 500 instruction should transfer control to the location 2500, and that is what precisely happens, if the base register B is loaded with a number 2000. It can be observed that a program with a length (core locations) less than or equal to $2^k$, (k= number of bits in the displacement field) can be relocated with one base register, and every $2^k$ size block of program would require one additional base register.

### 11.4.7. Stack Addressing

An instruction referring to the operands in a stack does not have any address field. Instead, the opcode itself decides that the operands for the instruction reside in a stack. A stack consists of data items stored in consecutive locations in the memory (may be main memory or a bank of registers). Associated with the stack is a pointer (address register) called Stack Pointer (SP) The stack pointer always points to the most recent (last) entry in the stack *i.e.*, top of the stack as shown in Fig. 11.4. Whenever a new item (data) is to be put in the stack, the stack

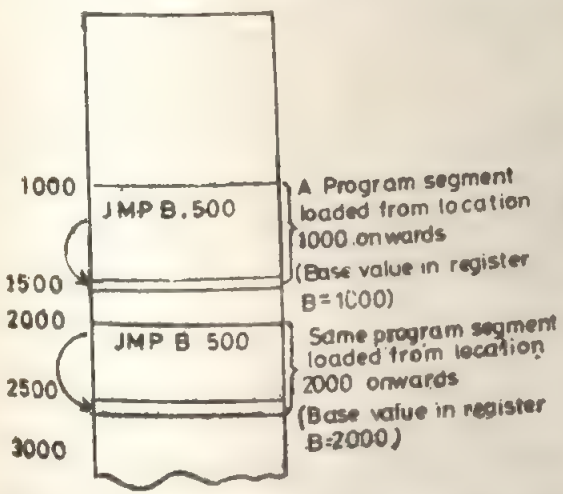



FIG 11.3 ILLUSTRATION OF BASE ADDRESSING

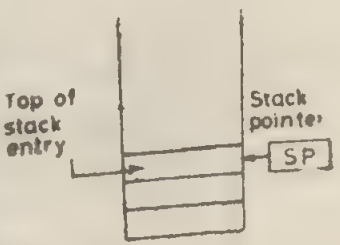



FIG 11.4 STACK

pointer (SP) is incremented by one and data is stored in the location indicated by the stack pointer. This operation is usually called as PUSH operation. On the other hand when a data item is removed from stack, the SP is decremented by one. This is called a POP operation. With these operations the stack acts as a Last In First Out (LIFO) device.

Stacks are useful in a number of applications. The most important of these are the implementation of subroutine jumps and evaluation of arithmetic expressions.

### Use of Stacks in Subroutines

A subroutine (subprogram) is a program written to carry out a specific task. This subprogram is called from the another (main)

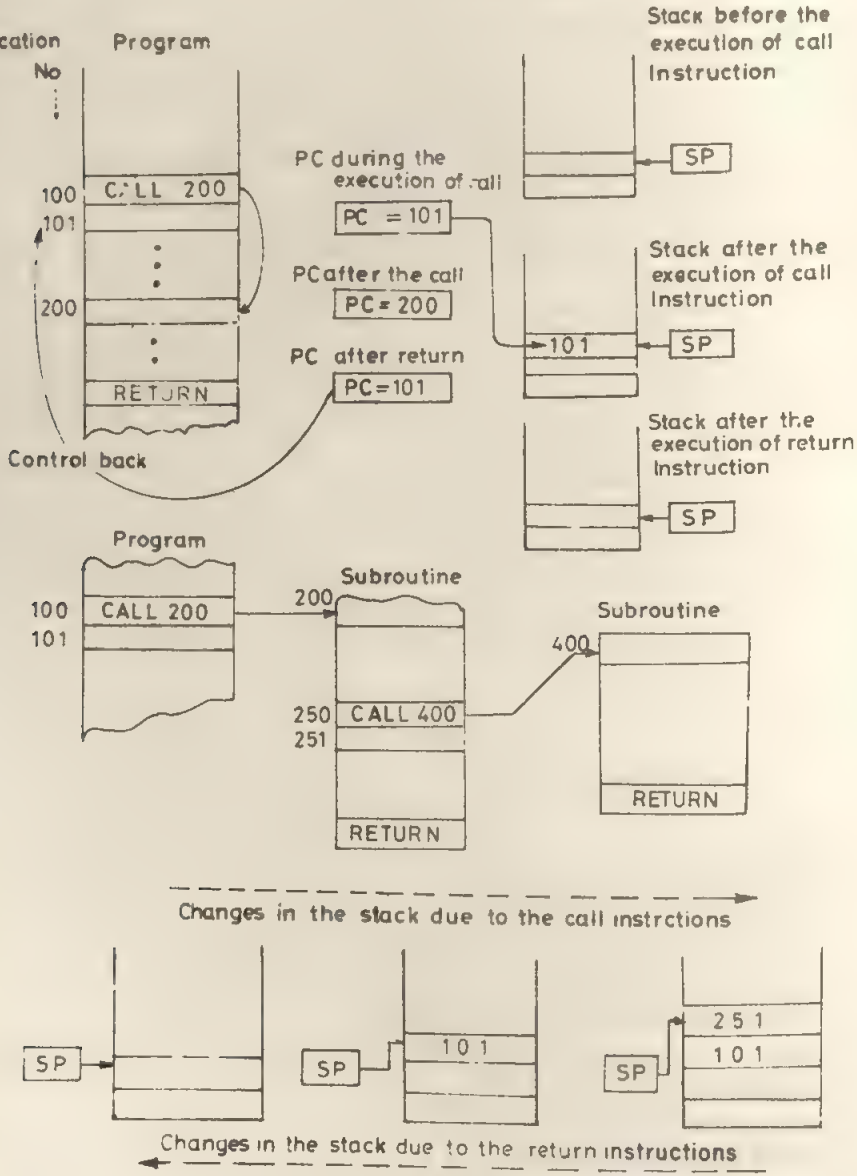



FIG 11·5 USE OF STACK IN CALL AND RETURN INSTRUCTIONS

program by the call instruction (jump to subroutine). The call instruction pushes the program counter on the stack and then passes control to the subroutine. The subroutine is executed, and the last instruction executed in the subroutine POPS the top of stack into PC thus causing control to go back to the main program at the appropriate place, *i.e.*, next instruction after the call instruction. (refer Fig. 11.5)

**Use of Stacks in the Evaluation of Expressions**

Another important use of stacks is in the evaluation of arithmetic expressions. The expression is scanned by the scanner program and operand/operators encountered are pushed on the stacks with the following rule.

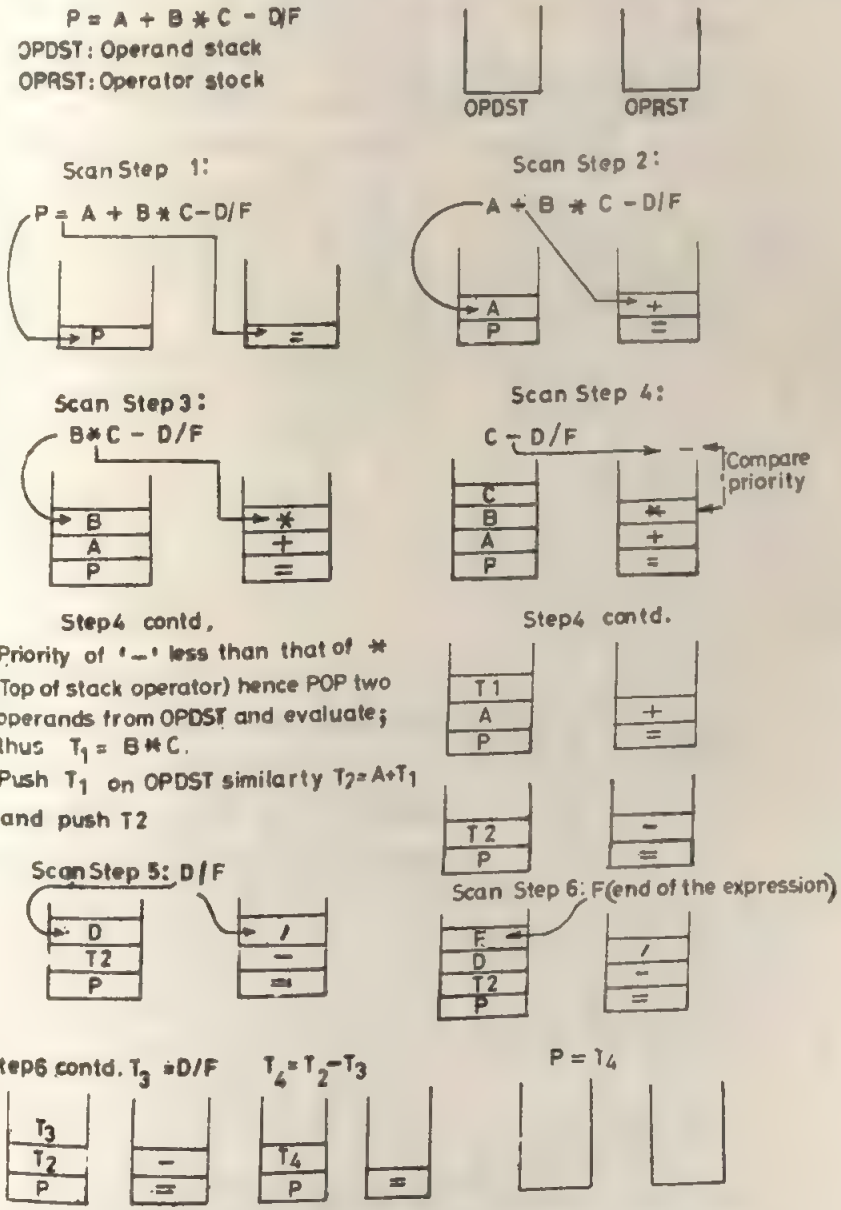



FIG.11·6 USE OF STACKS IN EVALUATION OF EXPRESSIONS

(We use two stacks one for operands and another for operators). The expression is scanned till an operator is found. This will give out operand and operator pair in one scan step. The operand is pushed on the operand stack, and the operator's (let us call it as an incoming operator) priority is compared with that on the top of operator stack. Depending upon the priority one of the following actions are performed.

(*a*) If the priority of the incoming operator is higher than that of the operator on the top of stack, the incoming operator is pushed on to the stack.

(*b*) If the priority of incoming operator is not higher (*i.e.*, lower or equal) than that on the top of the stack operator, then the top of stack operator from operator stack and the top two operands from the operand stack are poped and operation is evaluated. The result of the operation is pushed on the operand stack and the control is transferred back to the step (*a*). We illustrate this in Fig. 11.6 by showing the steps encountered in the evaluation of the expression,

$$P=A+B\times C-D/F$$

Priorities of . (multiplication) and / (division) operaters are assumed to be higher (this is the normal convention) than that of + (addition) or — (subtraction). Moreover, and / have same priority, so have + and —.

The stacks are invariably used for translating the arithmetic expressions (in higher level languages like FORTRAN etc.) into the sequence of basic machine language instructions. If stack addressing is not available at the machine language level, it could be simulated by software.

### 11.4.8. Immediate Addressing

In the immediate operand addressing mode, the operand field in the instruction format gives the operand itself instead of its address. Since the operand itself is a part of the instruction it is available (immediately) when the instruction is fetched. Though the length of the immediate operand is small due to the limited number of bits in the operand field, nevertheless a small operand is useful in many cases, like initialisation to 0, or use of a constant with a small value. For example, an instruction ADI 01 (add immediate 1 to accumulator) can be used to increment the accumulator. Similar uses of immediate addressing are possible.

### 11.4.9. Augmented Addressing

In this addressing mode, the number of bits in the operand address field is not sufficient to address every memory location directly. The operand address is formed by appending the bits from some register (usually program counter) to the address field. For example, in Hewlett Packard (HP) 2100A minicomputer there are 10 bits in the operand field, while the addressible memory is 32k words. Thus, we would require a 15-bit final address. The 5 bits of address is formed by attaching 5 most significant bits of the program counter (PC) to 10 bits in the operand address field as shown in Fig. 11.7.

We may consider the 5 bits as specifying one of the 32 blocks (called pages) where each block has $2^{10}=1024$ locations. The operand address field (10 bits) specifies the location with in a block. Since 5 most significant bits of the address are taken from the program counter, the location referred falls in the block in which the instruction (referring the operand) resides. With this mechanism an instruction cannot refer to a location in another page. In HP2100A, there is one additional bit (10th bit) in the instruction format which specifies whether 5 bits of PC should be used or 00000 should be used to form the final address. In the first case a location from a current page (page in which the instruction resides) is referred, while in the second case a location from $0^{th}$ page (called base page) is referred. Since there

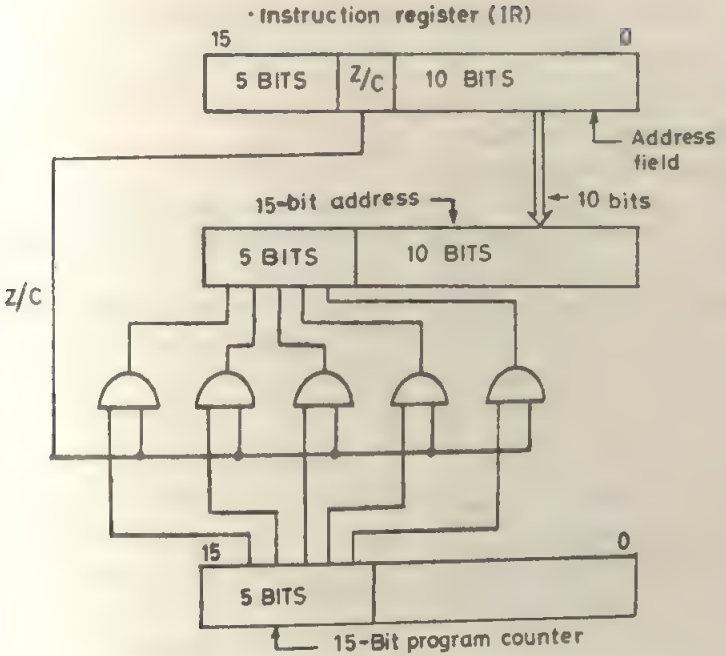



FIG. 11.7 ADDRESS FORMATION IN HP2100A MINICOMPUTER

exists indirect addressing also in HP2100A, the references to the operands in the other pages could be achieved through indirect addressing with pointers stored in the current or base page.

### 11.4.10. Implicit Addressing

In implicit addressing, the opcode of the instruction implicitly tells about the operand. For example, we may have an opcode which operates on some fixed registers. A more common example of implicit operand addressing is the use of the accumulator as one of the operands. For example, the ADD M instruction may specify the addition of the contents of the location M to the accumulator.

We have discussed most of the major addressing techniques, and with this background, we hope the reader will not find any difficulty in understanding the addressing modes of any particular machine. A machine may have one or more of the above addressing modes, possibly with slight variations.

## 11.5. INSTRUCTION SET OF A GENERAL PURPOSE COMPUTER

Machine languages of computers allow certain basic operations to be coded. Conventional machine language instructions can be separated into two groups, viz., (*i*) general purpose and (*ii*) special purpose. The general purpose instructions have wide applications while special purpose instructions are applicable for the specific application for which the machine is designed. A computer having a general purpose instruction set is called a general purpose computer. In this section, we shall study the various instructions which are available on a general purpose computer. The discussion is broken into a number of subsections, each subsection discussing a specific class of instructions.

### 11.5.1. Data Movement Instructions

The operation of copying data from one place to another is one of the most fundamental of all operations. We cannot have any useful computations without having any

data movement facility. Data movement instructions can be grouped as follows :

(*a*) Loading registers from main memory.
(*b*) Storing registers into main memory.
(*c*) Moving data from one register to another register.
(*d*) Moving data from one memory area to another.

All of the above four groups of operations may or may not be available on every machine. But the operations specified in (*a*) and (*b*) can be found in almost all the computers (in small computers these may be the only data movement instructions). While, all four groups may be available on a large machine. For example, IBM 360/370 series of machines have all four groups of data movement instructions. While HP2100A minicomputer has only first two groups.

### 11.5.2. Binary (Dyadic) Operation Instructions

All digital computers are equipped with an instruction to add two integers. Except for very inexpensive minicomputers, instructions for subtraction, multiplication and division of integers are standard as well. It is presumably, unnecessary to explain why computers are equipped with instructions to carry out arithmetic operations !

Another group of dyadic operations include Boolean function instructions. Although, there are 16 Boolean functions of two variables, very few computers (if any) have instructions to code all of them. But it is very rare to find a machine which does not have instructions for AND, OR and XOR operations. When we talk about a logical operation on operands, we mean a bit by bit logical operation on the two multi-bit operands. We shall discuss in the following paragraphs the use of logical instructions. A logical AND instruction is mainly useful in extracting a required part of a data word. For example, consider a 16-bit word storing two 8-bit characters. By ANDing this word with 0000 0000 1111 1111 we get the character occupying the least significant position (and 0's in the most significant position of the result).

The OR operation is used to merge the data of two operands. For example, we may read one character from a device into the accumulator. This could be shifted by 8 bits and new data byte could be OR*ed* with the earlier shifted data, thus giving the compact storage of the two data bytes. The use of XOR is mainly for checking the equality of two operands. This is easily possible since the result of XOR operation on two operands shall be 0, if and only if the two operands are equal. Ofcourse, the XOR instruction can also be used for merging data as is done by OR instruction.

### 11.5.3. Unary (Monadic) Operation Instructions

An unary operator operates on only one operand. Since monadic instructions require only one operand, they could be coded with smaller number of bits than dyadic instructions. In many cases the operand taking part in the monadic operations resides in the accumulator. The following instruction types with unary operations are commonly used in many digital computers.

(*a*) Instructions to shift the binary data. These are quite useful and are provided with many variations (like arithmetic/logic shifts, shifting with or without carry etc.)
(*b*) Instruction to clear accumulator.
(*c*) Instruction to increment/decrement a register (or memory location).
(*d*) An instruction to produce a 2's complement of an operand (usually in a register).
(*e*) Instructions which convert data from one form into another. The following conversions are commonly required.
  (*i*) Binary to BCD and vice-versa.

(*ii*) Converting BCD numbers to external (printable) form vice-versa.

(*iii*) Code translation (for example ASCII to EBCDIC etc.)

(*iv*) Fixed point to floating point and vice-versa.

### 11.5.4. Comparison And Branch (Jump) Instructions

The usefulness of comparison of two operands and alteration of flow in a program can be explained by a very simple example of computation. Consider a sample program which evaluates the square root of a number. If the operand is negative, program will be unable to find the square root (or it may behave in an unpredictable manner). If the machine has a facility to check whether the number is positive or negative, the program can have two branches, in one, it evaluates the square root (+ve operand) and in the other, it gives the error message.

At machine language level, we have various branch instructions. These instructions can change the program counter to a new value (branch address) specified in the address field of the instructions unconditionally or under some condition. Unconditional and conditional branch instructions are always found on any general purpose computer. Most common test conditions (for conditional branching) are zero, minus, plus and overflow of the result of some operation. In some machines, there are individual flip-flops (Flags) to store these conditions. One or more of these flip-flops are set or cleared as a result of the execution of an instruction (some instructions may not affect any of these flags).

### 11.5.5. Procedure call (subroutine jump) Instructions

A procedure (subroutine) is a program written to perform some specific task and it could be called (invoked) from several places in some other program (called main program). To facilitate this, processors are provided with 'Call' instructions (jump to subroutine instructions). The call instruction stores the program counter value in some known (to subroutine) memory location (may be stack) and executes a branch to the subroutine start address. The subroutine is executed and at the end, a RETURN (go back) instruction is executed, which places back the previously stored PC value into program counter, thus facilitating the return of the program control to point from which the subroutine was invoked.

### 11.5.6. Loop control Instructions

The need for a group of instructions (a program segment) to be executed several times (looping) occurs very frequently. Of course, we can implement loop control using conditional branch instructions and other instructions, however some machines provide machine language instructions to effect the loop control. These involve decrementing a counter every time a loop is executed and testing for '0' in the counter. Some machines also provide the automatic incrementation of a specified index register and the loop is terminated when a specified conditions is met.

### 11.5.7. Input/Output Instructions

These instructions are provided to transmit data/programs from the user of the machine to the memory of the computer and vice-versa. The user supplies data on some kind of medium (paper tape, punched cards, teleprinter key board etc., which the machine can read by executing input instructions. Similarly the machine outputs the data on the output medium (printer paper, punched cards etc.) by executing output instructions. The input/output instructions and their capabilities vary with the machines.

### 11.5.8. Machine control Instructions

This group comprises of instructions affecting machine states. The most common instruction is HALT which stops the program execution, some machines have states to

distinguish between the user mode and the system's mode. Also, in this group we have instructions to enable/disable interrupts.

## 11.6. INTERRUPTS AND TRAPS

Interrupts are the changes in the flow of a program, not carried out by the program itself, but by some external event. Interrupts are usually associated with I/O operations. For example, a program may have an instruction to start an I/O operation and set the device in a condition such that it causes the interrupt after the completion of the I/O operation. The interrupt mechanism in CPU after executing every machine instruction checks for the interrupt condition (by hardware as a part of an instruction execution cycle) and if it finds the interrupt ON, the instruction from a fixed memory location (instead of the usual next instruction in the sequence) is fetched and executed without incrementing or changing the program counter. The fixed location contains a jump to subroutine instruction and thus flow of program is transferred to the subroutine called the service routine. Service routine is usually written to carry out the next I/O operation. Of course, the machine status and other information pertaining to the interrupted program is to be stored just after the subroutine is entered and restored just before the execution of the return instruction. A CPU may be equipped to handle number of interrupt conditions generated by various devices.

The interrupt conditions just discussed were from the events occurring external to the CPU. On the other hand traps are conditions in the CPU itself causing the interrupts. For example, a machine may be equipped to interrupt when an OVF condition is generated in the ALU. The trap may be serviced by the service routine which may give a message to the user.

## EXERCISES

1. A computer has 4K main memory and 16-bit word length.

   (*a*) How many bits are required in the MAR.

   (*b*) Suppose the instruction format has 12-bits for direct address, how many different single address instructions can be coded in a 16-bit instruction word.

2. Is it possible to assign the expanding opcodes to allow all the following, to be coded in a 12-bit instruction format.

   An address field is 4-bit long.

   (*i*) 12 instructions with double address.

   (*ii*) 80 instructions with single address.

   (*iii*) 150 instructions with no address.

   If your answer is no, then calculate the maximum number of instructions possible in each of the address format, assuming that there must be 12 double address instructions.

3. Design an expanding opcode instruction format (with minimum number of bits) for a machine which provides the following facilities (all of these to be coded).

   (*i*) 15 general purpose registers

   (*ii*) 26 double address instructions with address 1 as register address, while address 2 is as follows.

   Address 2 has PC relative and index (any of the 15 registers) can be used as an index register addressing, with a provision to enable/disable the relative and index addressing.

(*iii*) 93 single address instructions with address 2 as an operand address.

Give the instruction format with various fields/subfields shown clearly.

In addition to the above instructions, how many no-address instructions can be additionally coded in the instruction format ?

4. A certain machine has 20-bit instructions and 8-bit addresses. If there are k ($k \leqslant 16$) double address instructions, what is the maximum number of single address instructions.

5. Is it possible to design as expanding opcode instruction format to allow all the following to be coded in a 32-bit instruction word.

3 instructions with two 15-bit addresses.
6000 instructions with one 15-bit address,
1 million instructions with no address.

If your answer is yes then give the instruction format and find out how many instructions are actually possible to code.

6. Base register addressing involves two lengths.

L1 : length of a field specifying a base register

L2 : length of a field specifying a displacement.

(*a*) How many distinct memory locations can be referred without changing the base register.

7. Devise a sequence of instructions to interchange the values of two variable X and Y without using a third variable or register (Hint : use XOR instruction).

8. State true or false :

(*a*) base addressing

(*i*) makes program relocatable

(*ii*) is same as index addressing

(*iii*) makes execution time of an instruction smaller

(*b*) Indirect addressing can be used to address data items in a table sequentially.

(*c*) Address calculations in base and index addressing are exactly same.

(*d*) Stack instructions have NO address

(*e*) Stack instructions have TWO addresses.

(*f*) Is direct addressing of entire memory always better than address formation assisted by other means (like indexing, base addressing, etc). Give your argument to support your answer.

9. What are the limitations regarding program relocation of the base addressing. On what does the size of program which could be relocated depend ?

If the program to be relocated is bigger in size than the maximum size, what remedies you suggest.

10. Suppose certain machine do not have a JSB (jump to Subroutine) instruction in the machine language, can it be simulated using other instructions. Give the sequence of instructions that could do the job.

11. Load register and store register instructions can carry out the data movement from one memory area to another, what advantage will be gained if a machine has memory to memory move instruction ?

12. Why do machines have logical instructions ? Give reasons with examples.

13. Certain machine does not have an instruction to clear the accumulater. How can you simulate this operation (give a sequence of other instructions).

14. Suppose a machine has no addition instruction, but there exists an instruction which can increment/decrement a register/memory location. How will you implement the addition operation on two positive numbers ?

15. Why do machines require interrupts and traps ?

16. What extra activities CPU will have to carry out in the instruction cycle (steps required) for the execution of an instruction, if interrupt facility is to be made available on a computer ?

□

# 12

# PROCESSOR DESIGN PRINCIPLES

## 12.1. INTRODUCTION

A PROCESSING unit is the main component of a computing system, around which other facilities exist. The basic task of a processor is to execute its machine language program resident in the main memory of the computer. The processor has two main components, *viz.*, (*i*) Arithmetic Logic Unit (ALU) and (*ii*) Control Unit (CU). An arithmetic logic unit could be designed using the principles discussed in the previous chapters on arithmetic. This chapter thus is devoted mainly to the design of control unit and the overall design of the information transfer and control mechanisms in the machine. A control unit could be designed using any of the two approaches, *viz.*, (*i*) Conventional (Hard-wired) Approach or (*ii*) Microprogrammed Approach.

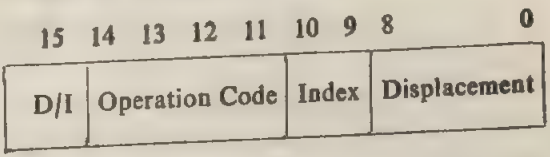
A processor, being a complex digital system, its design principles would be best illustrated through an example. We define a small hypothetical machine named as HM630 whose control unit design is discussed in this chapter.

## 12.2. HYPOTHETICAL MACHINE HM630

The block diagram of a machine under discussion is shown in Fig. 12.1. This configuration is typical of any digital computer with one processor.

### 12.2.1. Instruction Format of HM630

We assume that the HM630 has a 16-bit word length and all its instructions are 16 bit long. The instruction format is as follows :




There are three addressing modes in HM630, *viz.*, direct, indirect and indexed and we can address upto 32K (1K=1024) memory locations. In the direct address mode, ($15^{th}$ bit=0) the index field bits (bits 9 and 10), are 0's, and the binary number coded in bits 0 through 9 forms a direct address. When an index register is specified in the index field, the contents of the specified register (out of registers R1, R2 and R3) are added to the displacement bits. This forms an indexed address, and assuming that the $15^{th}$ bit is 0 (no indirect addressing), the location in main memory having the above address is accessed. The $15^{th}$ bit is used for specifying the indirect addressing. Indirect addressing has further indirection if the most significant bit of the addressed location contains a 1. In the following discussion the term effective address denotes the final address obtained as per the addressing mechanism just discussed.

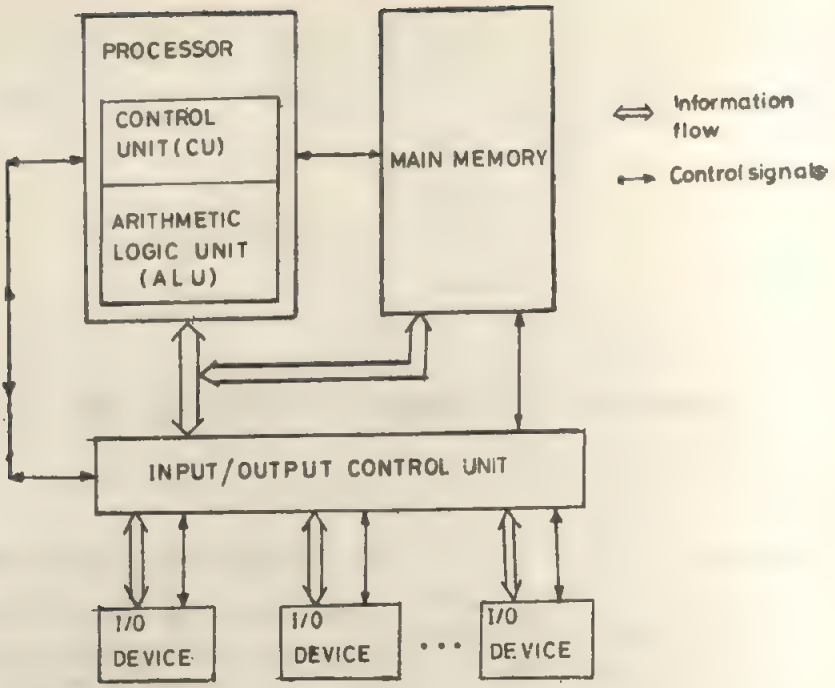



FIG. 12·1 ORGANISATION OF A DIGITAL COMPUTER

### 12.2.2 The Instruction Set

The instruction set of HM630 has 16 instructions which are discussed in this section

**Load Index Register (LIR)**

This instruction loads the contents of the memory location specified by the effective address into index register specified by the index field. Note that addressing for this instruction does not have indexing facility, since index field is used to specify the destination register for loading the data from the memory.

**Load the Accumulator (LDA)**

This instruction loads the contents of memory location specified by the effective address into the accumulator.

**Store the Accumulator (STA)**

This instruction causes the contents of accumulator to be stored in the memory location given by the effective address.

**Jump (JMP)**

This instruction alters the flow of program to the new address specified by the effective address, *i.e.*, effective address is loaded into the program counter (PC).

**Jump if Zero (JMPZ)**

This instruction executes the jump to the effective address if the result of last operation is zero.

**Jump if Negative (JMPN)**

It executes a jump to the effective address if the result of last operation is negative.

**Jump if Positive (JMPP)**

If the result of last operation is positive, this instruction executes a jump to the location specified by the effective address.

**Jump to Subroutine (JSB)**

This instruction stores the program counter in the location specified by the effective address and loads PC with the

address that is one more than the effective address. For example, in JSB M, where M is the effective address, the program counter (PC) is stored in location M and PC is loaded with the data M+1.

**Add to Accumulator (ADA)**

The contents of the memory location specified by the effective address is added to the contents of accumulator (AC) and the result is stored in the accumulator.

**Exclusive OR with AC (XOR)**

The contents of accumulator and contents of the location specified by the effective address are bit by bit Exclusive ORed and the result is placed in AC.

**AND with Accumulator (AND)**

The contents of the accumulator and the contents of the memory location specified by the effective address are ANDed bit by bit and the result is placed in AC.

**OR with Accumulator (OR)**

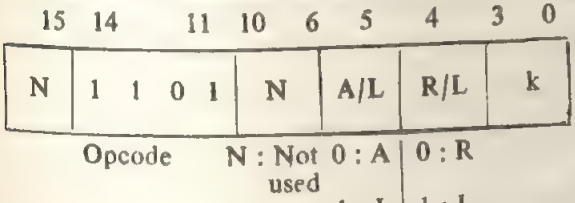
The contents of the accumulator and the contents of memory location specified by the effective address are ORed bit by bit and the result is placed in the accumulator.

**Complement Accumulator (CMA)**

The contents of the accumulator are complemented (2's complement).

**Shift Instructions (SH)**

The instruction format of the shift instruction is shown as follows :

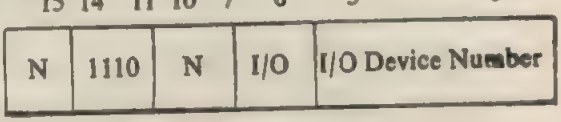



The number *k* specifies the number of positions in shift operation. The 4th bit indicates the direction of shift, *i.e.*, right or left shift while the 5th bit indicates the type of shift operation, *i.e.*, logical or arithmetical.

The instruction carries out the shift on the contents of the accumulator as indicated by the shift instruction format. (Note that K=0 means shift by 16)

**Input/Output (IO)**

The format of this instruction is shown below :




This instruction performs the input/output operation between the accumulater and the specified I/O device.

**Halt (HLT)**

This instructions stops the CPU, halting the execution of a program.

Table 12.1 gives the summary of the HM630 instructions.

## 12.3 ORGANISATION

Various units of the computer communicate to each other through the control signals and data paths. From this communication point of view, we shall describe the various functional units of HM630 in this section.

### 12.3.1. Memory

Fig. 12.2 shows the memory organisation as seen by the control unit. The Memory Buffer Register (MBR) holds the data during the access to the memory, while the Memory Address Register (MAR) holds the address of the location. The control signals read, write and clear, carry out the functions which are as follows :

**Read** signal when applied, reads the contents of memory location specified by the MAR (It initiates the read cycle of the memory) into the MBR.

**Write** signal when applied causes the contents of MBR to be written into the memory location. (It initiates the write cycle of the Memory).

TABLE 12.1

**HM630 Instructions**

| *Opcode* | *Mnemonic Name* | *Addressing Modes* | *Action* |
|---|---|---|---|
| 0000 | LIR | D/I | r←(EA) |
| 0001 | LDA | D/I, Index | AC←(EA) |
| 0010 | STA | D/I, Index | (EA)←(AC) |
| 0011 | JMP | D/I, Index | PC←EA |
| 0100 | JMPZ | D/I, Index | If (AC)=0 then PC←EA |
| 0101 | JMPN | D/I, Index | If (AC) <0 then PC←EA |
| 0110 | JMPP | D/I, Index | If (AC)> 0 then PC←EA |
| 0111 | JSB | D/I, Index | (EA)←(PC), PC←EA+1 |
| 1000 | ADA | D/I, Index | AC←(AC)+(EA) |
| 1001 | XOR | D/I, Index | AC←(AC)⊕(EA) |
| 1010 | AND | D/I, Index | AC←(AC) AND (EA) |
| 1011 | OR | D/I, Index | AC←(AC) OR (EA) |
| 1100 | CMA | D/I, Index | AC←($\overline{AC}$) |
| 1101 | SH | | Shifts AC |
| 1110 | I/0 | | Input/Output |
| 1111 | HLT | | Halts the CPU |

Notes : EA denotes the effective address

(EA) denotes the contents of memory location specified by EA

(PC) and (AC) denote the contents of PC and AC respectively.

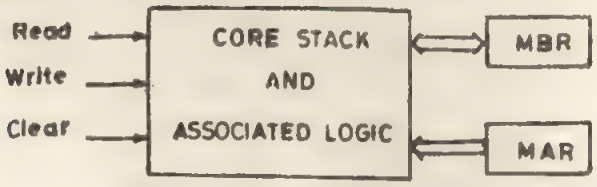



FIG. 12.2 MAIN MEMORY BLOCK DIAGRAM

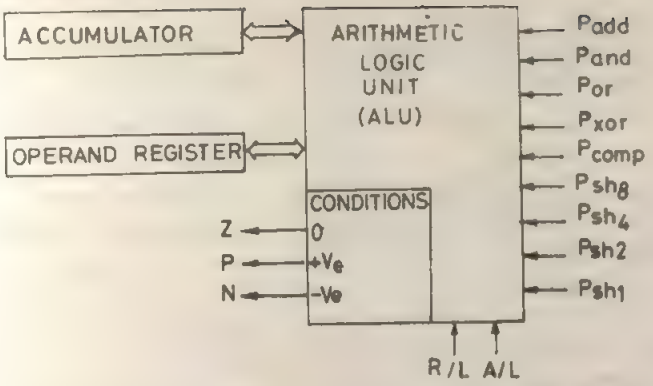



FIG. 12·3 ALU BLOCK DIAGRAM

**Clear** signal has same function as that of the read signal except that data read is not loaded into the MBR.

### 12.3.2. Arithmetic Logic Unit (ALU)

The arithmetic logic unit of HM630 could be designed using the principles discussed in the chapters on arithmetic. We assume that the ALU carries out its functions as per the control signals given by the control unit. Fig. 12.3 shows the ALU with control signals. The accumulator and the operand register (OR) hold the two operands and after an operation, the result is placed in the accumulator. The lines A/L and R/L indicates the type of shift operation. These lines shall be connected to the $5^{th}$ and $4^{th}$ bit of IR respectively. The lines Z, P and N indicates the condition of accumulator. The control lines $P_{add}$, $P_{and}$ etc., indicate the operations to be performed by ALU.

### 12.3.3. Input/Output Interface

This unit comprises of a common I/O controller which communicates with the specified I/O device through its interface. The I/O controller routes the data and control signal to an addressed peripheral (I/O) device interface. The peripheral interface unit block diagram is shown in Fig. 12.4.

The important units of I/O interface are Device Buffer, device flag and device control. Device Buffer is the buffer register of the device which holds the data to be inputed or outputed. Device flag signal indicates the completion of the I/O operation which was initiated earlier, by the start device signal.

## 12 4. CONVENTIONAL (HARD-WIRED) DESIGN OF THE CONTROL UNIT

The control unit has to supply the required signals to get the operations performed by various functional units as required by the stream of instructions (program) automatically.

This involves fetching the current instruction (indicated by PC), executing its

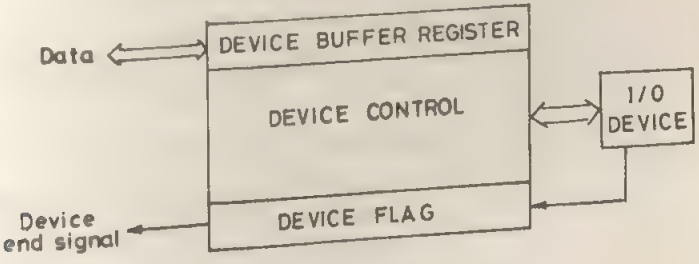



FIG. 12·4 I/O INTERFACE

operation and incrementing the PC and repeating these steps indefinitely.

A sequence of steps required for the execution of an instruction is called as, **Instruction cycle.** An **instruction cycle**. may be divided into a number of small cycles (called **machine cycles**), such that each machine cycle may carry out a specific subtask of the instruction cycle. Each machine cycle in turn may be broken into the number of steps or **micro-operations** (*i.e.*, an operation which could be carried out by a singie pulse) such that each step does a specific subtask of a particular machine cycle.

### 12.4.1. Control Unit Components

To execute an instruction, control unit fetches the instruction from main memory and decodes it. After decoding an operand if required, is fetched and the execution is carried out. To carry out these tasks, some registers and other elements are required in the control unit. These are discussed in the next paragraphs.

**Instruction Register (IR)** holds the instruction being executed.

**Operation Decoder** decodes the operation code and produces an active signal on one of the output lines.

**Pulse Sequencer (Pulse Distributor)** produces the pulses required in each machine cycle and provides the logic to sequence the machine cycles as per the instruction being executed.

**Program Counter (PC)** holds the address of the instruction to be executed. This register is also called as Instruction Address Register (IAR) or Instruction Address Counter (IAC).

**Transmission Bus** provides hardware for data transfer between any two registers.

The detailed block diagram of HM630 machines is shown in Fig. 12.5.

### 12.4.2. Discussion Of Machine Cycles

We identify the machine cycles (phases)

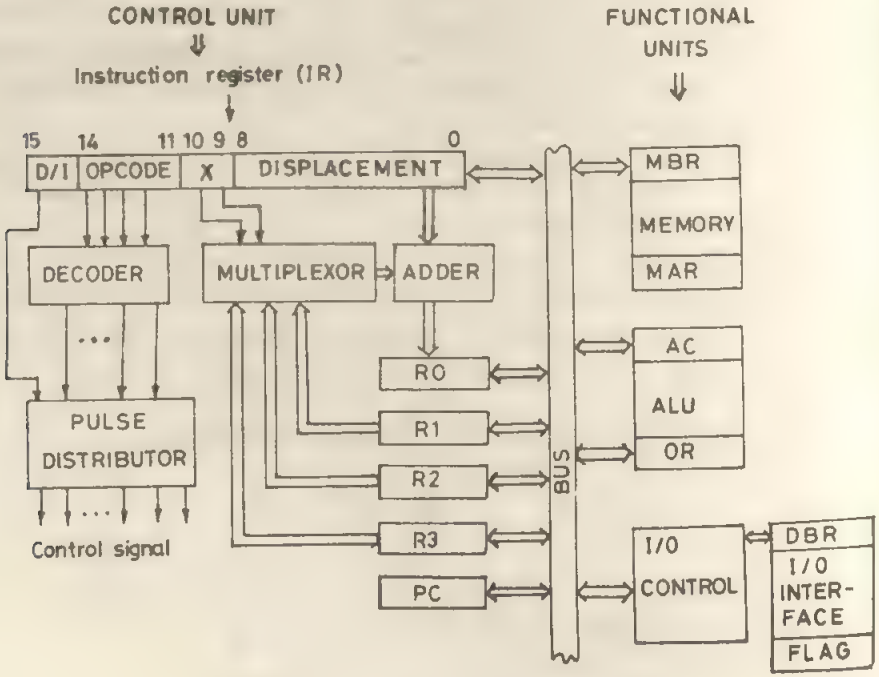



FIG. 12·5 HM 630 BLOCK DIAGRAM

required for HM630. These are selected on the basis of the activities to be carried out. An instruction cycle for a particular instruction may consists of one or more of these machine cycles in be appropriate sequence. To simplify the design of the sequencer we shall have all machine cycles with fixed number of steps.

**Machine Cycle 1 : Fetch Instruction (FI)**

The task of this machine cycle is to bring the instruction from the main memory into the instruction register and increment the program counter so that next time, the next instruction in sequence will be fetched. The steps (pulses) required to carry out the above task are listed below in order of the time sequence in which the indicated operations should be carried out. The steps are written as $P_{ij}$, where the number $i$ gives the machine cycle number while the number $j$ indicates the step number.

$P_{11}$ : Transfers contents of PC to MAR.
$P_{12}$ : Gives Read signal to the memory.
$P_{13}$ : No operation (gives time to the main memory to complete the Read Cycle).
$P_{14}$ : Transfers contents of MBR to IR.
$P_{15}$ : Adds the displacement to the specified index register and puts the result in RO, Increments the program counter; transfers the control to the next machine cycle.

**Machine Cycle 2 : Fetch Operand (FO)**

The task of this machine cycle is to bring the operand from main memory into the operand register (or any other register). The Address of the operand is assumed to be present in the register RO. The details of steps and their sequence is as follows :

$P_{21}$ : Transfers RO to MAR.
$P_{22}$ : Gives read signal to the memory.
$P_{23}$ : No operation.
$P_{24}$ : Transfers the contents of MBR to OR. Gives write signal to memory.
$P_{25}$ : Transfers the control to the next machine cycle.

**Machine Cycle 3 : Indirect Address Cycle (IND)**

This machine cycle resolves the indirect address and leaves the final address in the register RO. The steps are as follows :

$P_{31}$ : Transfers the contents of $R_0$ to MAR.
$P_{32}$ : Gives memory read signal.
$P_{33}$ : No operation.
$P_{34}$ : Transfer the contents of MBR to RO
$P_{35}$ : If $15^{th}$ bit of RO is 0 then transfers the control to the next machine cycle, else repeats IND machine cycle.

**Machine Cycle 4 : Special Cycle (SP)**

This machine cycle is provided for use in special cases. One or more pulses from this cycle could be used as per requirements. There shall also be five pulses in this cycle.

**Machine Cycle 5 : Store Cycle (ST)**

This machine cycle is used for recording the data from a register in the memory. The steps are as follows :

$P_{51}$ : Transfers RO to MAR.
$P_{52}$ : Gives clear signal to the memory.
$P_{53}$ : Transfers the contents of a register (whose contents are to be stored) to the MBR.
$P_{54}$ : Gives write signal to the memory.
$P_{55}$ : Transfers control to the next machine cycle.

The micro-operations in various machine cycles as stated are typical, and additions or deletions of the tasks could be made to any of the steps.

To implement the instruction set of HM630, machine cycles have to be made available to an instruction cycle as per the instruction being exempted. An instruction cycle would use the pulses from machine

cycles to generate the required control signals for the execution of the instruction.

### 12.4.3. Instruction Cycles for HM630 Instructions

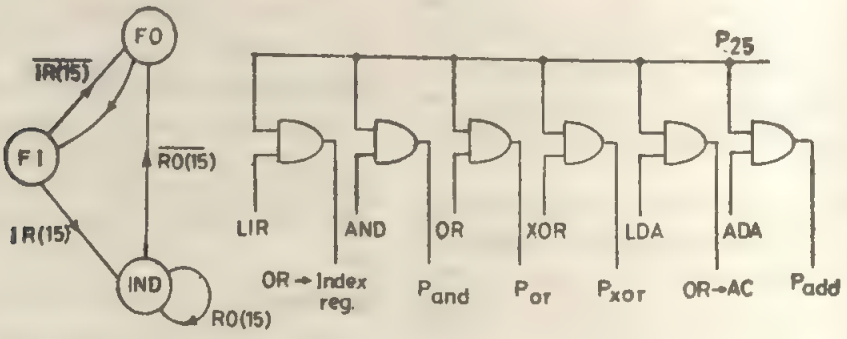
In this section the use of machine cycles to form an instruction cycle is discussed for various instructions of HM630. For this purpose we group instruction according to the micro-operations present in their instruction cycles.

**Group 1 (G1)** : The main characteristics of this group is that it requires an operand from the main memory. Thus all instructions which require memory data are grouped in group G1.

$$G1 = ADA + XOR + OR + AND + LIR + LDA$$

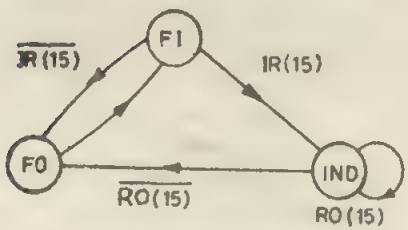
Where ADA, XOR, OR, AND, LIR, LDA are the symbolic names given to the opcode decoder output signals corresponding to these instructions respectively.

The machine cycles required and their sequence is given by the state diagram shown in Fig. 12.6 (*a*). The labels on directed edges indicate the condition under which the machine cycle transitions occur for the instruction in group G1. The FI machine cycle fetches the instruction and carries out indexing (except for the instruction LIR) and

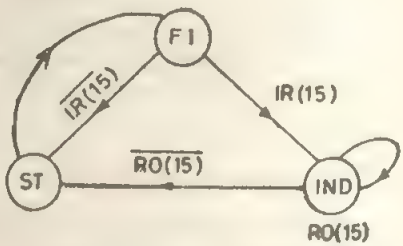



(a) $G_1$ Machine cycle switching

(b) $G_1$ Control signals

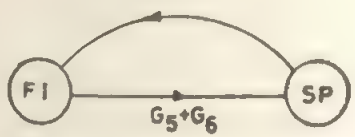



(c) $G_2$ Machine cycle switching




(d) $G_3$ Machine cycle switching


(e) $G_4$ Machine cycle switching




(f) $G_5$, $G_6$ Machine cycle switching

FIG. 12-6 MACHINE CYCLE LOGIC AND CONTROL SIGNALS

transfers the control to IND machine cycle if the indirect addressing is specified in the instruction [IR (15)=1], otherwise, the control goes to FO machine cycle. Also after the indirect address is resolved the control goes to machine cycle FO. Microoperations to be carried out in IND and FO machine cycles for this group are same as those discussed, except the pulse $P_{55}$ of FO machine cycle. This pulse is assigned the additional task of supplying a control signal to carry out the execution of the operation of an instruction. These controls are shown in Fig. 12.6 (*b*).

**Group (G2)** : This group consists of jump instructions, *i.e.*, JMP, JMPP, JMPN and JMPZ instructions. Thus signal G2 is obtained by ORing the decoded outputs of codes of JMP, JMPP, JMPN and JMPZ, *i.e.*,

$$G2 = JMP + JMPN + JMPP + JMPZ.$$

The machine cycles used are FI, FO and optionally IND. The machine cycle state diagram is shown in Fig. 12.6 (*c*) for this group. The microoperations in the machine cycles FI and IND are as per the discussion of group G1. The FO machine cycle is used to carry out the execution of jump *i.e.*, contents of RO (effective address) is transferred to PC as per the conditions. This requires only one pulse from FO machine cycle. We use pulse $P_{21}$ for this job, while the other pulses from this cycle should not do their normal operations of reading the memory since G2 instructions do not require an operand. For example read signal is given by $P_{23}.\overline{G_2}$. The logic for the signal (say PT) which transfers RO to PC is given by :

$$P_T = P_{21}. (JMP + JMPN . N + JMPP . P + JMPZ . Z)$$

where N, P and Z are negative and zero condition signals in ALU.

**Group (G3)** : This group of instructions includes instructions which involve the storage of data in the main memory. In HM630 we have only two instructions, *i.e.*, JSB and STA which require a store cycle. Thus :

$$G3 = JSB + STA$$

In this group, JSB stores the contents of PC while STA stores the contents of AC in the memory location. Therefore, the machine cycles required are FI, ST and optionally IND. The machine cycle transition diagram for this group is shown in Fig. 12.6 (*d*).

The micro operations in FI and IND machine cycles are as per our earlier general discussion of machine cycles. The steps in ST cycle are given as follows :

**ST Cycle Details for Group G3**

$P_{51}$ : Transfers the contents of RO to MAR.

$P_{52}$ : Gives clear signal to memory.

$P_{53}$ : If the instruction being executed is JSB then transfers the contents of PC to MBR. If the instruction being executed is STA then transfers contents of AC to MBR.

$P_{54}$ : Gives write signal to memory. If the instruction being executed is JSB then transfers the contents of RO to PC.

$P_{55}$ : If the instruction being executed is JSB then increments the register PC by one. Transfers control to FI cycle.

**Group 4 (G4)** : This group has two instructions *viz.*, HLT and CMA . This group require only one machine cycle *i.e.*, FI. The micro operations in FI machine cycle are same, as those discussed earlier except the pulse $P_{15}$. It has the following task :

$P_{15}$ : If the instruction being executed is HLT, it stops the CPU clock. If the instruction being executed is CMA, it complements the contents of AC.

Fig. 12.6 (*e*) shows the machine cycle transition diagram for this group. Since HLT or CMA instruction requires only one machine cycle, *i.e.*, FI, the machine cycle transition shall not occur in the present case.

**Group 5 (G5)** : This group is composed of the I/O instruction only. We use SP machine cycle to carry out the execution cycle of this

instruction. The machine cycle transition diagram is shown in Fig. 12.6 (*f*). The FI cycle microoperations are same as discussed earlier, except that $P_{15}$ will not carry out the indexing. The SP machine cycle is used as follows :

**Input ($6^{th}$ bit of IR=0)**

$P_{41}$ : NOP

$P_{42}$ : Gives the start device signal and stops the CPU clock

$P_{43}$ : NOP

$P_{44}$ : Transfers the data from Device Buffer (DB to AC)

$P_{45}$ : Transfers the control to machine cycle 1

**Output ($6^{th}$ bit of IR=1)**

$P_{41}$ : Transfers the contents of AC to Device Buffer (DB)

$P_{42}$ : Same as input

$P_{43}$ : NOP

$P_{44}$ : NOP

$P_{45}$ : Transfers the control to the machine cycle 1

**Note** : Since I/O devices are usually very slow compared to the CPU, when $P_{42}$ starts an I/O operation, it also stops the CPU clock. The clock is restarted by device flag signal.

**Group 6 (G6)** : This group has only shift instruction. The machine cycles used are FI and SP and their state diagram is shown in Fig. 12 6 (*f*).

The FI machine cycle is same as per our earlier discussion. In the SP machine cycle we have to carry out shift operation on the contents of the AC as per the bits 0 to 5 of the IR. The bits of the instruction register coding A/L and R/L operations are connected to the corresponding lines in ALU. The number of shifts is given by the 4-bit number coded in bits 0 to 3 of the IR, with 0000 combination giving a 16-bit shift. We decode this combination and call the decoded signal as Z, *i.e.*,

$$Z = \overline{IR(0)} \cdot \overline{IR(1)} \cdot \overline{IR(2)} \cdot \overline{IR(3)}$$

The SP cycle details are :

$P_{41}$ : If Z+IR(0)=1 then gives $P_{sA1}$ to ALU.

$P_{42}$ : If Z+IR(1)=1 then gives $P_{sA2}$ to ALU.

$P_{43}$ : If Z+IR(2)=1 then gives $P_{sA4}$ to ALU.

$P_{44}$ : If Z+IR(3 =1 then gives $P_{sA8}$ to ALU.

$P_{45}$ : If Z=1 then gives $P_{sA1}$ to ALU. Transfers the control to the machine cycle FI.

## 12.4. MACHINE CYCLE LOGIC

The pulses required for the five machine cycles can be derived from the logic shown in Fig. 12.7. The 5-bit circulating register produces five pulses in time sequence from a square wave oscillator (Main Clock). These 5 pulses are in turn gated with 5 flip-flops of five machine cycles. Whenever a machine cycle is required for an instruction, the corresponding machine cycle flip-flop is set. Initially, the circulating register contains 10000, while FI cycle flip-flop is set and the other cycle flip-flops are reset. This will start the machine from FI machine cycle. The Set—Rest logic of various machine cycle flip-flops is derived from the discussion, we had, for instruction cycles of HM630 machine instructions. The complete machine cycle transition diagram is shown in Fig. 12.8. In the machine cycle transition diagram the labels on arcs indicate the condition under which the transitions in machine cycles occur. An arrow indicates the direction of the transition. **The transitions are activated by the last ($5^{th}$) pulses of machine cycles as per the conditions shown on arcs.** From this state diagram, we can derive the Set—Rest logic expression for the flip-flops of the machine cycles. These expressions are :

FI set : System Reset+$P_{45}$+$P_{55}$+$P_{35}$

FI reset : $\overline{CMA+HLT} \cdot P_{15}$

FO set : $\overline{IR(15)} \cdot (G1+G2) \cdot P_{15}+\overline{RO(15)} \cdot (G1+G2) \cdot P_{35}$

FO reset=$P_{35}$+System Reset

IND set=IR(15) . (G1+G2+G3) . $P_{15}$

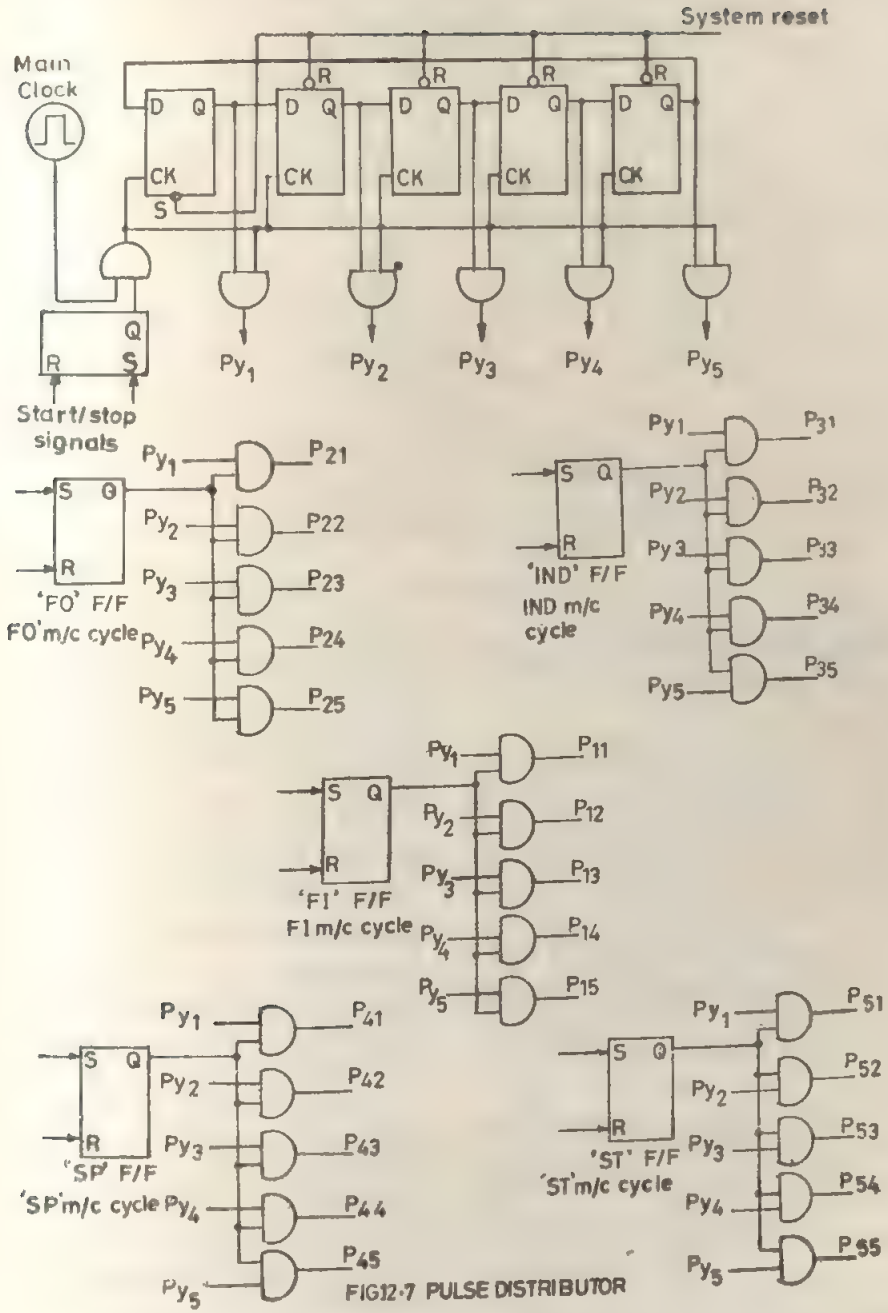



FIG12·7 PULSE DISTRIBUTOR

IND reset = $\overline{RO(15)} \cdot P_{85}$ + System Reset
SP set = $P_{15} \cdot (IO + SH)$
SP reset = $P_{45}$ + System Reset
ST set = $IR(\overline{15}) \cdot G3 \cdot P_{15} + \overline{RO(15)} \cdot G3 \cdot P_{85}$
ST reset = $P_{85}$

This completes the discussion of the nardwired design of the control unit of HM630. The next section discusses the micro-programmed design.

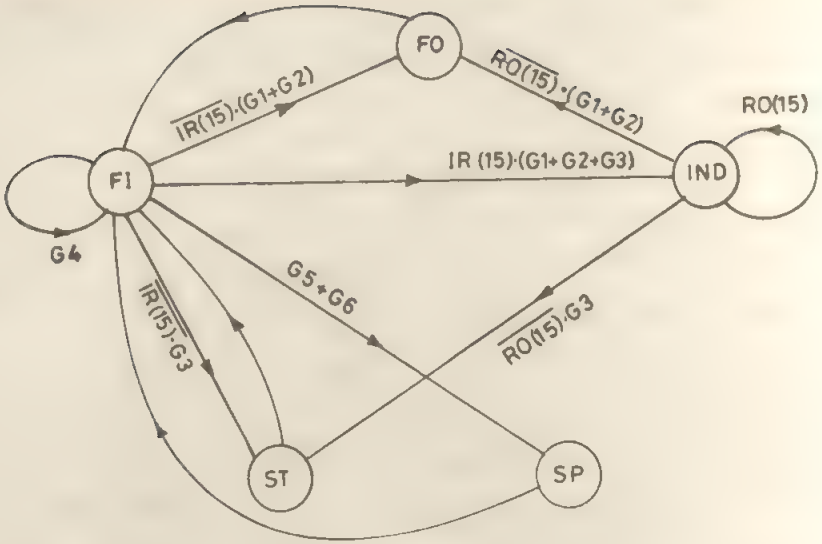



FIG.12·8 HM630 MACHINE CYCLE STATE DIAGRAM

## 12.5. MICROPROGRAMMED DESIGN OF CONTROL UNIT

In the design discussed in section 12.4, the machine language of HM630 is interpretted by various hardware units through the sequences of control signals which are generated by the hardwired machine cycle logic. In other words the machine language is directly interpretted by the logic circuits. In such a design any change in the instruction set thus would require changes in the control unit hardware. Wilkies (1951) suggested a new approach of microprogrammed control unit design. In this approach, a machine language is interpretted by an interpreter running in a lower level language (a language closer to the hardware). The lower level language has instructions called micro-instructions. Each **microinstruction** is a coded description of a set of microoperations.

Programs written using microinstructions are called **microprograms**. The memory where these programs reside is called the control memory. Since a microoperation can be executed in a small time, the control memory should match the microinstruction execution speed. In 50's and early 60's the memory technology was not very advanced and the idea of microprogram control was not very attractive. But the situation changed after late 60's, and with the advancement of semiconductor memory technology, most of the third generation computers were designed using this concept.

### 12.5.1. Basic Principles

The block diagram of a microprogrammed control unit of a CPU is shown in Fig. 12.9. In this figure Read only memory (ROM) stores microprograms required to interpret the machine language instructions. Each machine language instruction is interpretted by executing a microroutine which is written so as to carry out the instruction cycle of the machine language instruction. When a machine language instruction is fetched into the instruction register (IR), the opcode is translated into an address of the ROM location, from where the microroutine of the instruction is stored. In Fig. 12.9, the ROM address mapper generates the starting addresses for various instructions as per their opcodes. The complete machine instruction

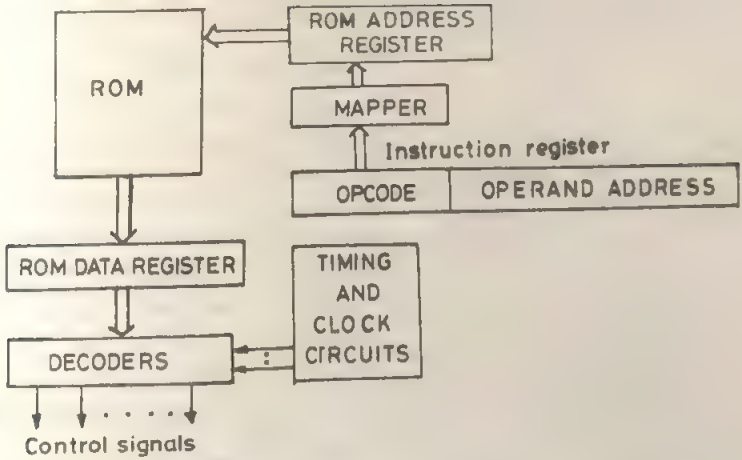



FIG. 12 9 CONTROL UNIT SCHEMATIC OF A MICROPROGRAMMED CPU

is executed as follows. When the machine is started, the control unit executes an instruction fetch microroutine which is common to all the machine instructions. This brings the machine instruction in the instruction register. At the end of this routine, the output of mapper is transferred to the ROM address register (RAR) (which acts as a microprogram counter). This transfers the microprogram control to the starting address of the microroutine of the machine instruction being executed, and the execution of the microprogram from this location continues. This in effect executes the microroutine of the machine language instruction. At the end, the microprogram control is transferred to the instruction fetch routine and the process repeats.

Microinstructions are executed as follows. A microinstruction pointed by RAR is read into ROM Data Register (RDR). The RDR has fields corresponding to the various groups of microoperations. The decoders for these groups decode the microoperations and generate the corresponding control signals which are used to activate various units.

To implement the HM630 instruction set using the above principles, we shall first decide on the format of microinstructions. A microinstruction should provide facility to code any of the required microoperations which we came across, while studying the conventional design of HM630 control unit in the earlier section. We shall divide these into groups and develop the format for specifying them in a microinstruction.

### 12.5.2. Bus Control Microoperations

Bus is used to carry out the data transfer between two registers (source and destination registers). To specify such microoperations, we therefore shall require two fields, one for specifying the address of the source register, while the other for specifying the destination register. The lengths (number of bits) of each of these fields should be sufficient to address any of the required registers. We shall name the source field as RTB (Register to Bus) and destination field as BTR (Bus to Register).

In HM630 we have 12 registers and hence we shall have 4 bits in each of these fields.

### 12.5 3. Function (FN) Group

Besides the above bus control microoperations, we require a field to specify microoperations, to provide control signals to the functional units (ALU, I/O, Memory) for

carrying out functions. These microoperations are listed below with their description.

**List of Function Group of Microoperations**

| *Code* | *Mnemonic Name* | *Description* |
|---|---|---|
| 0. | NOP | No operation |
| 1. | ADD | Gives $P_{add}$ pulse to ALU |
| 2. | AND | Gives $P_{and}$ pulse to ALU |
| 3. | OR | Gives $P_{or}$ pulse to ALU |
| 4. | XOR | Gives $P_{xor}$ pulse to ALU |
| 5. | SH8 | Shift control signals to ALU |
| 6. | SH4 | |
| 7. | SH2 | |
| 8. | SH1 | |
| 9. | READ | Read signal to memory |
| 10. | CLEAR | Clear signal to memory |
| 11. | WRITE | Write signal to memory |
| 12. | ADDI | Index register is added to displacement and result put in RO |
| 13. | INPUT | Transfers the data from device buffer to AC |
| 14. | OUTPUT | Transfer the data from AC to device buffer |
| 15. | CLF | Clear the device flag |
| 16. | STD | Start the device for operation |
| 17. | JMP | Jump in micro-program |

We have listed 17 microoperations and if more facilities at hardware level are made available microoperations for these could be defined. Since we have 17 microoperations, we require 5 bits in function field to specify any one of these 17 microoperations. Moreover we have enough space to extend the set if extensions to the set of microoperations is required.

### 12.5.3. Test Field

The microoperations in test field are required to alter the flow of microprogram depending on various conditions. Typical action of a microoperation of this group is to skip the next microinstruction if a condition specified by the microoperation is true, else if the condition specified is false, no operation is carried out. The list of these microoperations is as follows :

**Test Field Microoperations**

| *Code* | *Mnemonic Name* | *Action* |
|---|---|---|
| 0. | NOP | No operation |
| 1. | SKZ (ALU tests) | Skip on zero |
| 2. | SKN (ALU tests) | Skip on negative |
| 3. | SKP (ALU tests) | Skip on positive |
| 4. | SKIR 15 | Skip if IR (15)=0 |
| 5. | SKIR 0 | Skip if IR (0)=0 |
| 6. | SKIR 1 | Skip if IR (1)=0 |
| 7. | SKIR 2 | Skip if IR (2)=0 |
| 8. | SKIR 3 | Skip if IR (3)=0 |
| 9. | SKRO 15 | Skip if RO (15)=0 |
| 10. | SKF | Skip if device flag=1 |

Since there are 10 microoperations in this field, we shall require 4 bits to code any one of them. Additional 6 skip conditions if required could be coded.

### 12.5.4. Microinstruction Format

With the discussion we just had, the following format for HM630 microinstruction is suggested.


The last field is a special (SP) field with 3 bits. In FN field, if we code a JMP microoperation, we require to specify the jump address in microprogram. To avoid the additional bits in the microinstruction, we

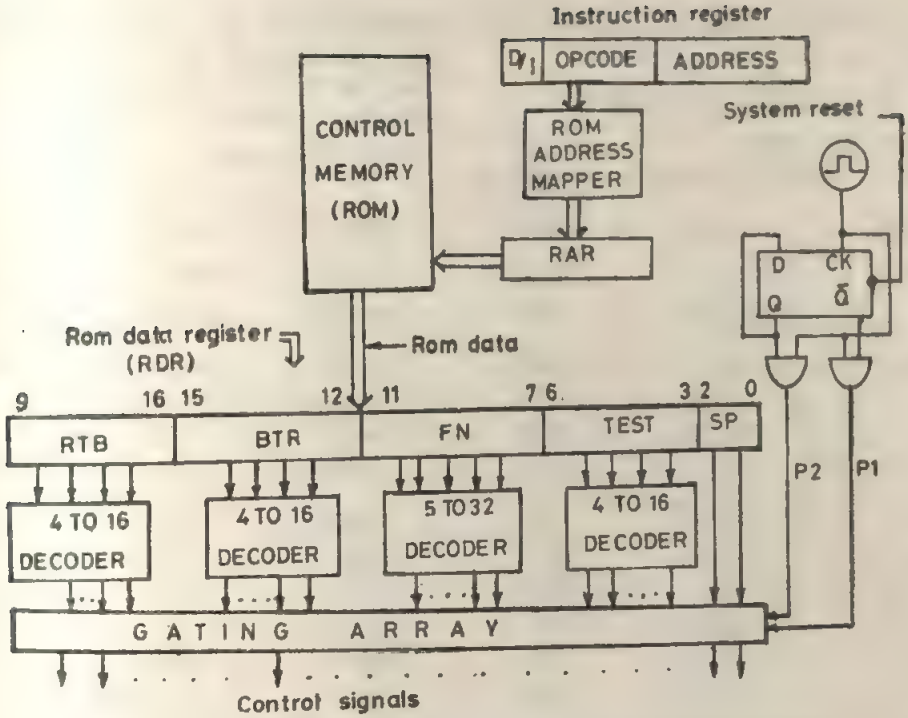



FIG. 12-10 MICROPROGRAMMED CONTROL UNIT SCHEMATIC OF HM630

shall use 4 bits in test field and 3 bits of SP field, to specify a jump address. This can address 128 locations of control memory which is sufficient for a small machine like HM630.

SP field bits can be used when JMP is not coded in FN field. We use 2 bits of SP field as follows for coding the additional 2 microoperations.

SP Field Details

| INCPC | MAP | Unused |
|---|---|---|
| 1 | 1 | 1 |

The INCPC microoperation increments the program counter of CPU by one while MAP microoperation transfers the output of the mapper to the ROM address register. This microoperation thus can be used to pass the microprogram control to the micro-routine of the machine language instruction to be executed.

The block diagram of the HM630 micro-program control unit is shown in Fig. 12.10.

In Fig. 12.10 ROM data register (RDR) holds the current microinstruction. The microinstruction fields are decoded and control signals are generated through the clock pulses $P_1$ and $P_2$. The microoperations in the first two fields are activated by clock pulse $P_1$, while the microoperations in the last three fields are activated by the clock pulse $P_2$ ($P_2$ comes after the pulse $P_1$). The clocks $P_1$ and $P_2$ can be very easily obtained as shown in Fig. 12 10

The additional register SP (Scratch Pad) is provided (not shown) for temporary storage of data in the microprogram. Particularly SP will be used to simulate the subroutine call operation using JMP microoperation. The data from ROM is loaded by the rising edge of $P_1$ whereas the falling edge of $P_1$ carries out the execution of the bus field microinstructions. Also $P_1$ increments the RAR so that the next

microinstruction in sequence is read from the control memory in the next cycle.

### 12.5.5. Sample Microprograms

This section discusses a few sample microprograms. The microroutines are written in symbolic language (microassembly). We give microroutines for fetching the instruction, the indirect address cycle and an execution routine for the ADA machine language instruction.

| *LABEL* | *RTB* | *BTR* | *FN* | *TEST* | *SP* |
|---|---|---|---|---|---|
| FETCH | PC | MAR | READ | NOP | INCPC |
| | MBR | IR | WRITE | NOP | MAP |
| IND | RO | MAR | READ | NOP | NOP |
| | MBR | RO | WRITE | SKRO15 | NOP |
| | NOP | NOP | JMP | —IND— | —IND— |
| | SP | RAR | NOP | NOP | NOP |
| ADA | NOP | NOP | ADDI | SKIR15 | NOP |
| | RAR | SP | JMP | —IND— | —IND— |
| | RO | MAR | READ | NOP | NOP |
| | MBR | OR | WRITE | NOP | NOP |
| | NOP | NOP | ADD | NOP | NOP |
| | NOP | NOP | JMP | —FETCH— | |

When the machine is turned on, the system reset signal sets RAR to point to the fetch routine address. When machine is started, the microprogram starts from the fetch routine. This brings the instruction in the instruction register. The MAP microoperation at the end of the FETCH routine transfers the control to the microroutine of the instruction present in the IR. Assuming that we have ADA instruction in the IR, the control will come to the label ADA. The first microinstruction of this routine carries out the indexing (ADDI microoperation) and checks for the indirect addressing (SKIR 15 microoperation). If indirect address is specified in the machine language instruction, the control goes to the next microinstruction which stores RAR (microprogram counter) in SP and causes a branch to the IND microroutine, otherwise this microinstruction is skipped. The next two microinstructions get the operand in OR. The add operation is performed next, and the last microinstruction transfers the control to the FETCH routine and the microprogram executes the next machine language instruction similarly.

### 12.5.6. Hardware Details

This section gives the logic circuit implementation of the microprogrammed control unit.

#### Bus Field Hardware

Fig. 12.11 shows the bus field logic. The bus lines are driven by tri-state drivers. A tri-state driver output has 3-states 0, 1 and high impedance.

The driver, when enabled by a control signal, connects (logically) its input data line to the bus line, otherwise (when disabled) the driver output is at high impedance thus disconnecting the bus from the driver. Fig. 12.11 shows only 1 bit of the bus, other 15 bits have identical logic.

The RTB and BTR field microoperations are executed as follows. The data from the source register is selected by enabling the drivers at the output of the source register. The enable signal is taken from the decoder of the RTB field. Thus the data from the source register appears at the Bus. The bus lines are the input lines (D inputs) of all the

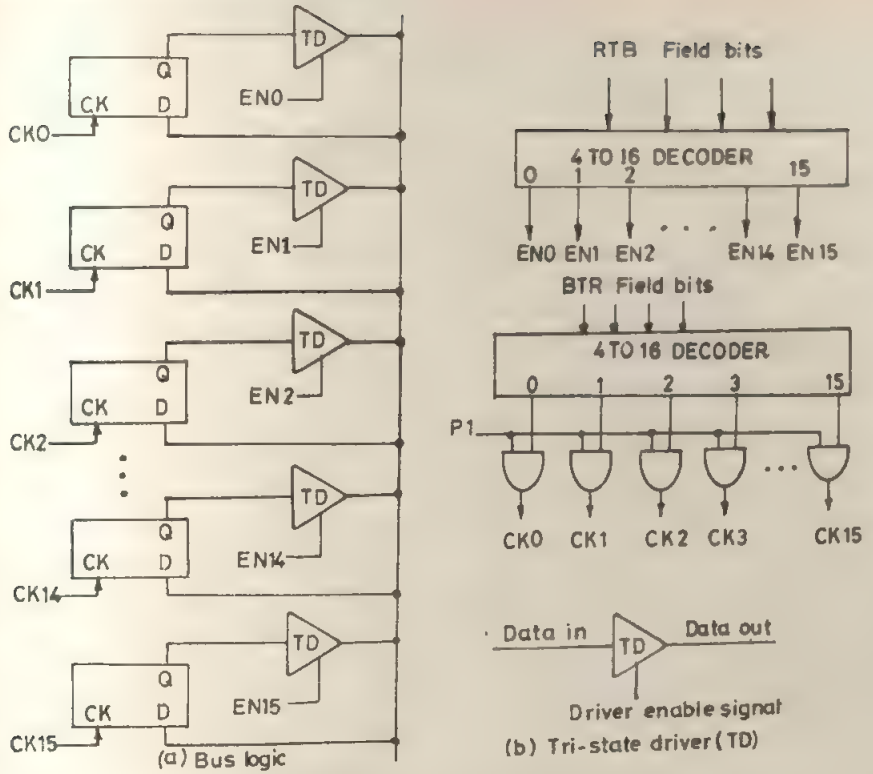



FIG. 12·11 LOGIC FOR BUS MICROOPERATIONS

registers. To select a destination register we just have to clock it. This clock $CK_n$ (for selecting $n^{th}$ register) is derived from BTR field decoder outputs and the pulse $P_1$.

### Function Field Hardware

The 5-bit function field is decoded by a 5 to 32 decoder as shown in Fig. 12.12. The control signals are derived from the pulse $P_2$ as the outputs of AND gates. These signals are given to the functional units to signal them for carrying out the various micro-operations.

### Test and SP Field Microoperations

The 4-bit test field is decoded by a 4 to 16 decoder as shown in Fig. 12.13. A micro-operation in this field, tests for a particular

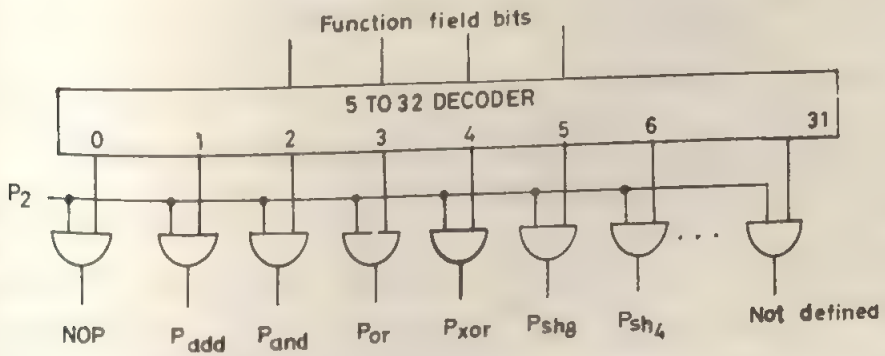
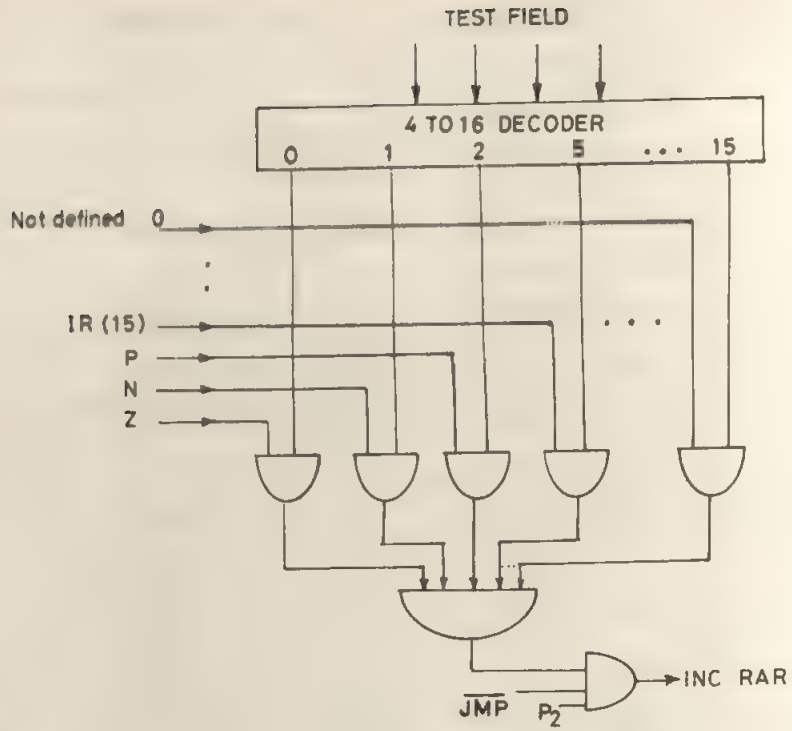
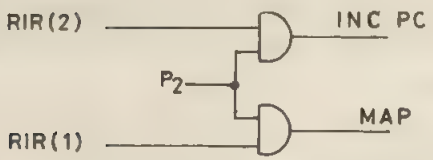



FIG. 12·12 FUNCTION FIELD CONTROL SIGNALS




(a) Logic for TEST field microoperations




(b) Logic for SP field microoperations

FIG. 12·13 LOGIC FOR TEST AND SP FIELDS

condition and if the condition is true the ROM address register is incremented by 1, which amounts to the skipping of the next microinstruction. In Fig. 12.13 (*a*), the output of OR gate produces the increment RAR signal. The JMP signal given to the AND gate inhibits the skip (as required), when the JMP is coded in the function field.

The hardware to generate control signals of the SP field is shown in Fig. 12.13(b). Since we have one bit each, for each individual microoperation in this field, we do not require a decoder for this group.

### 12.5.7. Microprogramming Applications

Although microprogramming was introduced as a systematic method of designing the control unit, its other implications are far reaching. By changing the ROM contents, one can change the instruction set of the machine. Moreover, using RAM as control

memory, new microprograms could be loaded in the control memory. This thus opens the possibility of having dynamically changing instruction sets. Many functions traditionally done by software could be implemented by microprograms. Language translators which run usually in machine languages, could be run partly or fully in a microprogramming language, thus speeding up the compilation process. Also the maintenance procedures (test programs in a machine language) could be microprogrammed. This will result in faster diagnostics with better resolution, because of the fact that microprograms directly interact with the hardware. The software written and run as microprogram is called **Firmware.**

A microinstruction format without any encoding (*i.e.*, 1 bit for one microoperation), is called **horizontal**, while the format having maximum encoding is called **vertical.** Strictly horizontal or vertical formats are uncommon and most machines use the composite approach.

## EXERCISES

1. With reference to the conventional control unit of HM630. Write down the switching expressions for the following signals :
   (*a*) Memory read signal.
   (*b*) A signal which transfers RO to PC.
   (*c*) A signal which carries out the indexing.
   (*d*) $P_{SM1}$ signal to ALU.
   (*e*) Padd signal to the ALU.
   (*f*) Memory write signal.
2. Repeat (1) for microprogrammed control unit.
3. Analyse the following microprograms (FETCH and IND routines as per the Section 12.5) and find HM630 language instructions they interpret.

| Label | RTB | BTR | FN | TEST | SP |
|---|---|---|---|---|---|
| (*a*) XYZ | NOP | NOP | ADDI | SKIR | 15 NOP |
| | RAR | SP | JMP | —IND—— | |
| | RO | MAR | READ | NOP | NOP |
| | PC | MBR | WRITE | NOP | NOP |
| | RO | PC | NOP | NOP | INCPC |
| | NOP | NOP | JMP | —FETCH | —— |
| (*b*) RT | NOP | NOP | ADDI | SKIR | 15 NOP |
| | RAR | SP | JMP | ——IND—— | |
| | NOP | NOP | NOP | SKN | NOP |
| | NOP | NOP | JMP | —FETCH— | |
| | RO | PC | JMP | —FETCH— | |

(*c*)

| | | | | |
|---|---|---|---|---|
| MM | NOP | NOP | ADDI | SKIR 15 NOP |
| | RAR | SP | JMP | ——IND—— |
| | RO | MAR | READ | NOP NOP |
| | AC | MBR | WRITE | NOP NOP |
| | NOP | NOP | JMP | —FETCH— |

4. Write the microprograms for the following :
   (*a*) Shift instruction of HM630.
   (*b*) To multiply the 8 bit numbers kept in AC and OR (assume additional microoperations in the appropriate fields if required).
   (*c*) A routine which converts the 8-bit binary number into 2-digit BCD number (assume the additional microoperations in the appropriate fields if necessary).
5. It is desired to include the instruction ISZ M (Increment the contents of location M by 1, and skip the next instruction if the result is 0) in the HM630. Give the pulse-wise description of machine cycles required in the instruction cycle of the ISZ M instruction.
6. With reference to the hardwired design of HM630 CPU, write the

logic expressions for :

(*a*) Set-Reset signals of all the machine cycle flip-flops

(*b*) Memory clear signal

(*c*) Increment PC signal

(*d*) Signal $P_{AA}$ to ALU

7. One of the machine cycles for group G2 is changed as follows :

   Instead of FO cycle (which was used earlier for carrying out the transfer of jump address), we use SP cycle, with this change, develop the machine cycle switching diagram and derive Set-Reset logic for all the machine cycle flip-flops.

8. Instruction DLD M (double load) which loads the contents of two consecutive locations from address M (effective address) in AC and OR is to be incorporated in HM630. Give the pulse-wise description of machine cycle required in the instruction cycle of this instruction.

9. Analyse the following microprograms and find out which machine language instruction of HM630 each of these interpret (assume FETCH routine as per the discussion given).

```
Label   RTB   BTR   FN     TEST      SP
(a) RTN NOP   NOP   ADDI   SKIR 15   NOP
        RAR   SP    JMP    —ADDCN—
        NOP   NOP   NOP    SKN       NOP
        NOP   NOP   JMP    —FETCH—
```

```
Label   RTB   BTR   FN     TEST      SP
ADDCN   RO    MAR   READ   NOP       NOP
        MBR   RO    WRITE  SKRO15    NOP
        NOP   NOP   JMP    —ADDCN—
        SP    RAR   NOP    NOP       NOP
(b) XYZ NOP   NOP   ADDI   SKIR15    NOP
        RAR   SP    JMP    —ADDCN—
        ROR   MAR   READ   NOP       NOP
        MBR   AC    WRITE  NOP       NOP
        NOP   NOP   JMP    —FETCH—
```

10. Write microprograms for HM630 to interpret :

    (*a*) DLD instruction (exercise 8)

    (*b*) ISZ instruction (exercise 5)

    (*c*) STA instruction of HM630

11. Write microprograms to interpret :

    (*a*) Comparison division algorithm

    (*b*) Booth's multiplication algorithm.

12. What are the various factors to be considered in designing the micro-instruction formats.

13. In the design of hardwired control unit, bus design was not discussed. Also a transfer operation was specified by a single pulse. Could you suggest how to get the source and destination (4-bit each) quantities ? (Hint : Use m/c state flip-flops as inputs.)

□

# 13

# INPUT/OUTPUT DEVICES AND SYSTEMS

## 13.1. INTRODUCTION

INPUT/Output (I/O) devices provide the means of information communication between the computer and the outer world. To carry out any computational work using a computer, data and instructions must be supplied to the computer and it must deliver the results. The perforated (punched) paper tapes were the first popular media for storing programs and data. Before computers were commercially available, the use of paper tape was well established in telegraph systems.

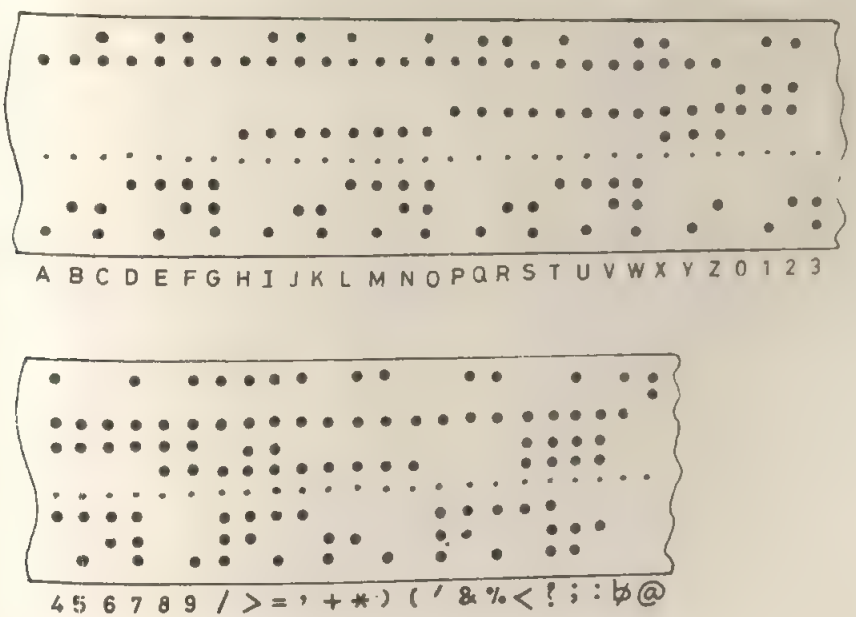



FIG. 13-1 A PERFORATED PAPER TAPE IN ASCII CODE

and devices for punching and reading these tapes were already well developed. Modern computers have card readers, paper tape readers, teleprinters, CRT terminals, line printers, card punchers and magnetic tapes and disks as their I/O devices.

## 13.2. PAPER TAPE DEVICES

Most of the present day computers use 8-track, one inch wide paper tapes. Data on the paper tape is punched by a punching machine. The codes representing information on tapes may vary from one computer to another, but most commonly used codes are ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). The 7-bit ASCII code uses 7-bits for information and the 8th bit is optionally used for the parity (this bit is 1 if the number of 1's in the information bits is odd, otherwise it is 0). The EBCDIC is an 8-bit code. The Table 13.1 gives the ASCII code while Table 13.2 gives the EBCDIC code for the alphanumeric information (English alphabets, numerals and other symbols). The Fig. 13.1 shows a perforated tape carrying the data in ASCII code which includes all the characters commonly encountered.

TABLE 13.1 THE ASCII CODE

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000 | NULL | DCO | BLANK | 0 | @ | P | | |
| 0001 | SOM | $DC_1$ | ! | 1 | A | Q | | |
| 0010 | EOA | $DC_2$ | " | 2 | B | R | | |
| 0011 | EOM | $DC_3$ | # | 3 | C | S | | |
| 0100 | EOT | $DC_4$ | $ | 4 | D | T | | |
| 0101 | WRU | ERR | % | 5 | E | U | | |
| 0110 | RU | SYNC | & | 6 | F | V | | |
| 0111 | BELL | LEM | ' | 7 | G | W | | |
| 1000 | $FE_0$ | $S_0$ | ( | 8 | H | X | | |
| 1001 | HT/SK | $S_1$ | ) | 9 | I | Y | | |
| 1010 | LF | $S_2$ | * | : | J | Z | | |
| 1011 | $V_{TAB}$ | $S_3$ | + | ; | K | [ | | |
| 1100 | FF | $S_4$ | , | < | L | \ | | ACK |
| 1101 | CR | $S_5$ | - | = | M | ] | | |
| 1110 | SO | $S_6$ | . | > | N | ↑ | | ESC |
| 1111 | SI | $S_7$ | / | ? | O | ← | | DEL |

Examples

Code for Q is 101 0001
Code for C is 100 0011

### The Abbreviations Used in Table 13.1

| | | | |
|---|---|---|---|
| NULL | Null (idle) | FF | Form Feed |
| SOM | Start of Message | CR | Carriage Return |
| EOA | End of Address | SO | Shift Out |
| EOT | End of Transmission | SI | Shift In |
| WRU | Who are you | $DC_0$—$DC_4$ | Device Control |
| RU | Are You | ERR | Error |
| BELL | Audible Signal | SYNC | Synchronous |
| FE | Format Effecter | LEM | Logical End of Media |
| HT | Horizontal Tabulation | $S_0$...$S_7$ | Separaters |
| SK | Skip | ACK | Acknowledge |
| LF | Line Feed | ESC | Escape |
| $V_{tab}$ | Vertical Tabulation | DEL | Delete |

TABLE 13-2 : THE EBCDIC CODE

| | 10 | | 10 | | 11 | | 11 | | 00 | | 00 | | 01 | | 01 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 |
| 0000 | | | | | (9) | (10) | (11) | (12) | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) |
| 0001 | a | j | | | A | J | (14) | 1 | | | SOS | | | | (13) | |
| 0010 | b | k | s | | B | K | S | 2 | | | FS | | | | | |
| 0011 | c | l | t | | C | L | T | 3 | | TM | | | | | | |
| 0100 | d | m | u | | D | M | U | 4 | FF | RES | BYP | PN | | | | |
| 0101 | e | n | v | | E | N | V | 5 | HT | NL | LF | RS | | | | |
| 0110 | f | o | w | | F | O | W | 6 | LC | BS | EOB | UC | | | | |
| 0111 | g | p | x | | G | P | X | 7 | DL | IL | PRE | EOT | | | | |
| 1000 | h | q | y | | H | Q | Y | 8 | | | | | | | | |
| 1001 | i | r | z | | I | R | Z | 9 | | | | | | | | |
| 1010 | | | | | | | | | | CC | SM | | ¢ | ! | (15) | : |
| 1011 | | | | | | | | | | | | | . | $ | , | # |
| 1100 | | | | | | | | | | | | | < | * | % | @ |
| 1101 | | | | | | | | | | | | | ( | ) | - | ' |
| 1110 | | | | | | | | | | | | | + | ; | > | = |
| 1111 | | | | | | | | | CU1 | CU2 | CU3 | | \| | ¬ | ? | " |

**The Abbreviations Used in Table 13.2**

| | | |
|---|---|---|
| PF | Punch Off | 1 * 12-0-9-8-1 |
| LC | Lower Case | 2 * 12-11-9-8-1 |
| DL | Delete | 3 * 11-0-9-8-1 |
| CU | Custmer Use | 4 * 12-11-0-9-8-1 |
| TM | Tape Mark | 5 * No punches |
| RES | Restore | 6 * 12 |
| NL | New Line | 7 * 11 |
| BS | Back Space | 8 * 12-11-0 |
| IL | Idle | 9 * 12-0 |
| CC | Cursor Control | 10 * 11-0 |
| DS | Digit Select | 11 * 0-8-2 |
| SOS | Start of Significance | 12 * 0 |
| FS | Field Spectator | 13 * 0-1 |
| BYP | Bypass | 14 * 11-0-9-1 |
| LF | Line Feed | 15 * 12-11 |
| EOB | End of Block | |
| PRE | Prefix | * Indicates the |
| SM | Set Mode | punchings on |
| PN | Punch On | rows given |
| RS | Reader Stop | against the key |
| UC | Upper Case | (Punched card |
| SP | Space | rows). |

**Example to illustrate the use of the Table 13.2**

| *Character* | *Code* | |
|---|---|---|
| A | 1100 | 0001 |
| 1 | 1111 | 0001 |
| p | 1001 | 0111 |
| 14 * | 11;0,9,1 | |

### 13.2.1. Paper Tape Reader

A paper tape reader consists of a reading station and a mechanism to move the tape under it whenever required. The reading station is composed of photo cells and a lamp assembly to produce a slit of light covering all the tracks. The presence or absence of a hole is indicated by a photo cell which produces two distinct voltages. These voltages are amplified and shaped by amplifier-shapers so that they represent the logical 1 or 0 signals. The amplifier-shaper (AS) used for synchro-track generates the clock pulse which latches the data from the eight amplifier-shapers into the 8-bit device buffer register. This data is then transmitted to the computer and the tape is moved for reading the next character.

The operational speeds of paper tape readers are expressed as the maximum number of characters which can be read per second. Present day paper tape readers operate upto the speeds of 2000 characters per second. Most of the photo tape readers are generally friction driven and they move the tape continuously till a stop condition is sensed. Extremely fast starting and braking of the tape are highly desirable features and most readers are capable of stopping on any given character.

A typical paper tape reader interface to the computer is shown in Fig. 13.2. It works as follows. The computer gives a command to the device which clears the device flag and moves the tape. This transmits a data byte (8 bits) into the device buffer register. The CPU checks for the flag of the device (which was cleared earlier when the command was given) and if the flag is ON, the data is taken by the CPU from the device buffer. On the other hand if flag is OFF, the CPU does not take the data, but either waits for the flag to becomes ON or does some other task.

Some paper tape readers use mechanical sensors for reading the data. These are very slow readers. In these readers the nine (one for SYNC track) teeth shaped sensors sense the holes (if there is a hole, tooth passes through it, otherwise it is blocked by the tape) and accordingly activates the levers, which finally activate the contact switches giving out the data read from the tape.

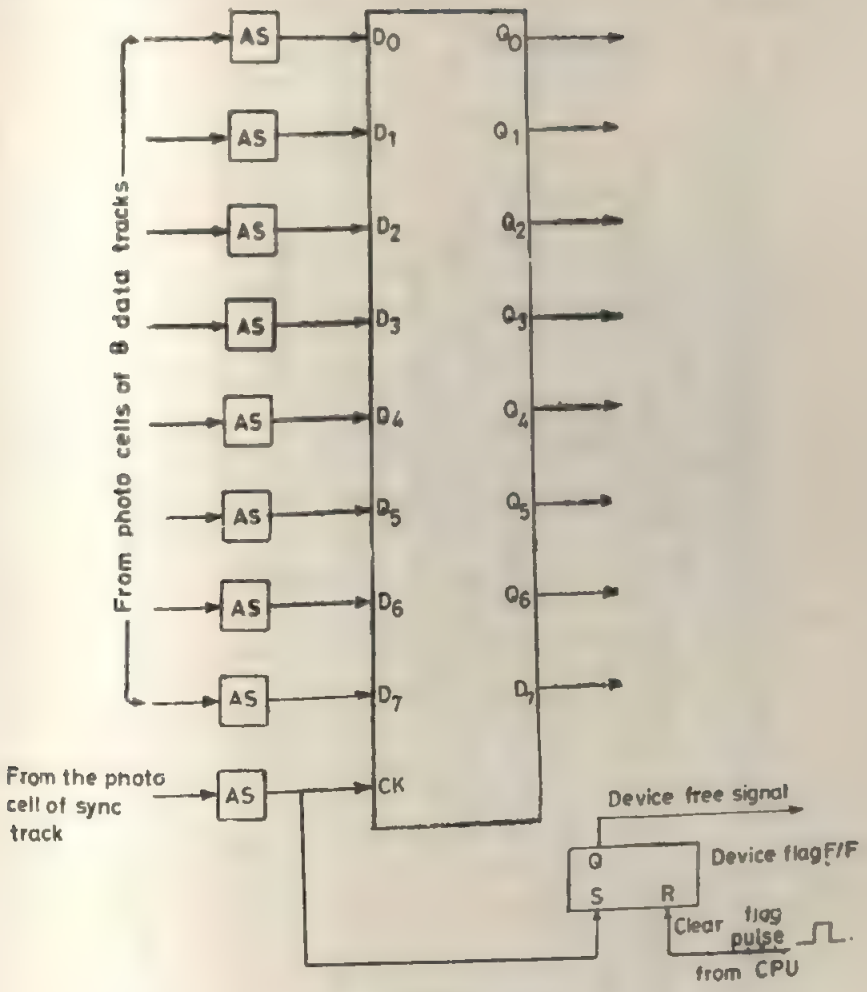



FIG.13·2 A TYPICAL PAPER TAPE READER INTERFACE

### 13.2.2. Paper Tape Punch

Paper tape punching devices use the mechanical tape punching mechanism (eight punchers for eight tracks and one special puncher for the synchro track. These punchers are activated by electromagnets which are energised (for certain duration required to punch a hole). Electromagnets are energised as per the bits in the data byte. For example, if the 3rd and 4th bit in the byte is ON, the third and fourth electromagnets are energised and holes in 3rd and 4th tracks are made. The computer interface for a paper tape punch consists of a device buffer register a flag flip-flop and some control logic. The data byte from computer is transferred to the device buffer logic and the computer issues a punch command and at the same time clears the device flag (to indicate the busy status of the device). The devices punches the data and moves the tape forward by one character position. The mechanism for tape movement consists of a motor and a tape spool with appropriate bracking and tape release mechanism.

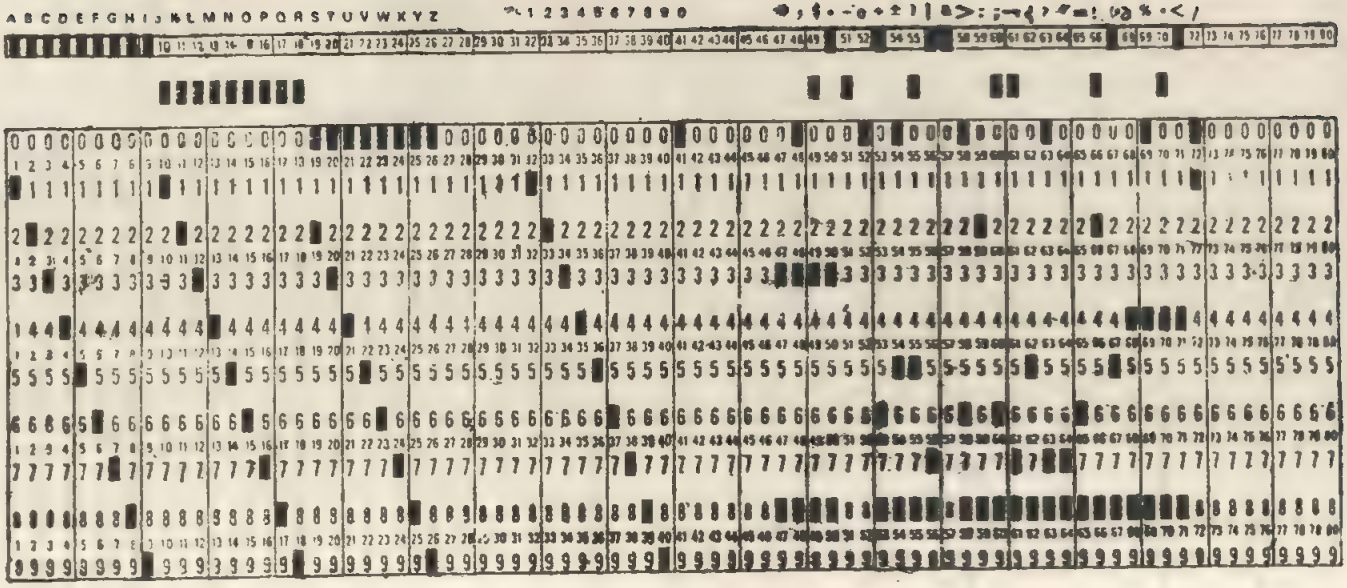


FIG. 13.3 A PUNCHED CARD

## 13.3. PUNCHED CARD DEVICES

Punched cards are the most widely used medium. There are a number of card sizes, but the most commonly used card at present is a 12 row, 80 column card having a size of $7\frac{3}{4} \times 3\frac{1}{4}$ square inches. Data on a punched card can be coded in a numerous ways, but the most commonly used code is the Hollerith code. This is an alphanumeric code in which character code is punched in a single column having 12 positions (rows). Fig. 13.3 shows a punched card with Hollerith codes (hole combinations) for various characters. For example, the code for character A has punchings on the top row and row 1. Note that rows (except the top two rows) bear their numbers in all the 80 columns in the commercially available cards (Fig. 13.3).

Data on the punched cards is prepared by a card punch machine (key punch machine). A key punch machine has an hopper to keep the deck of blank (without data) cards. One card at a time is fed to the card punch station and a key punch operator types in the data through the type-writer like key board and thus the data is punched on the card.

### 13.3.1. Card Readers

Card reading equipment have a read station made of 12 photo cells and a light slit arranged along a column of the card. The card is passed through the reading station. The punchings on each column (all 12 rows) are read by photo cell and amplifier shaper assembly, and the 12-bit data from each column is transmitted to the computer one after another. Some card reade. have logic to convert the Hollerith code of the punched character into the standard 8-bit code (ASCII or EBCDIC). The deck of punched cards is kept in a hopper of the card reader and an electromechanical mechanism upon the read command pushes a bottom most card from the hopper into the read station. The card passes through the reading station and is collected in the collecting hopper. In this processes, data on the entire card is transmitted to the computer memory.

## 13.4. L. PRINTERS

A line printer prints one line of the information in one print cycle. Electromechanical printers are commonly used and have printing speeds varying from 200 lines to 1500 lines per minute. Printing mechanism consists of a drum (chain in chain type printers) having mirror images of letters engraved on the circumference as shown in Fig. 13.4. The drum is kept rotating at

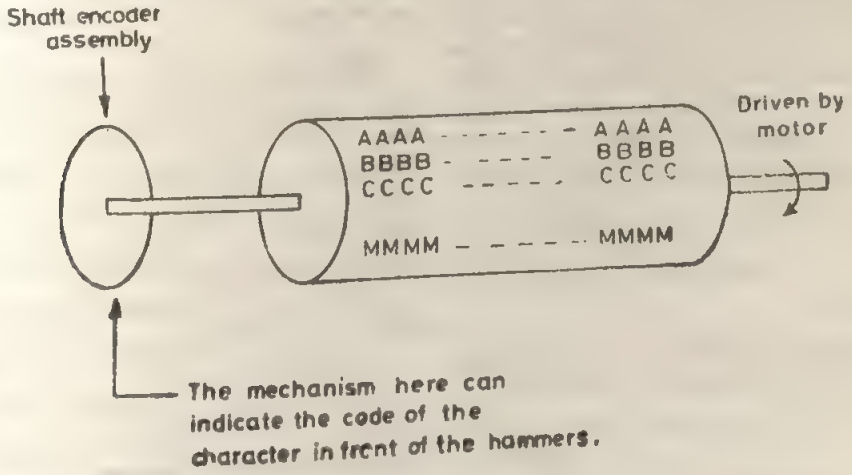



FIG 13·4 LINE PRINTER DRUM

certain revolutions per minute (This decides the printing speed). The characters forming a line to be printed are transmitted from computer to the local line memory in the printer. The drum, while rotating, positions each row of the characters (A's, B's...etc.) along the hammers (which hit the paper, behind which the carbon ribon and engraved character row lie). Which engraved line of characters is in front of the printing position is determined by the shaft encoder attached to the rotating shaft of the drum. The hammers are energised for all character positions in which there exist a match between the character indicated by the shaft encoder (character facing the hammers) and the characters in the printer line memory. After one revolution of drum, all characters in the line buffer would have been printed, thus giving the maximum printing speed of one line per revolution of the drum.

## 13.5. TELEPRINTER (TTY) AND CRT TERMINALS

Teleprinter (TTY) and CRT terminals are commonly used devices in digital communication systems. Also, in a time-shared computer system, where a number of users are served simultaneously by the system, users communicate to the system through TTY or CRT terminals. A TTY or CRT terminal has a type-writer like key-board through which user types his data (or program) into the system. This data is also printed on the TTY printer (or displayed on CRT). Since TTY or CRT terminals are communication devices, they are designed with a standard serial communication interface, through which data is received or transmitted in a serial bit by bit manner.

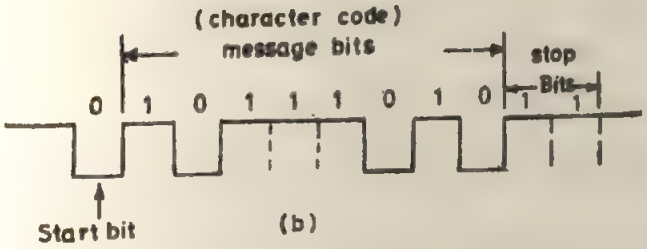
Fig. 13.5 shows the format of a serial data byte. The $T_s$ and $R_s$ lines are the transmit and receive lines through which the terminal transmits or receives the data respectively. When no communication is being done, both the $R_s$ and $T_s$ lines are at logical 1 state. Computer interface for a TTY or CRT terminal consists of an 11-bit shift register and some control lines. An 8-bit character to be printed (or displayed) on the terminal is loaded in the shift register with the start and stop bit values as shown in Fig. 13.5 (c). The data in the shift register is shifted out and is received and printed (displayed) by the device.

The driver shown in Fig. 13.5 (d) produces a 12 V (20 mA) and line open conditions for logical 1 and 0 TTL signals respectively at the input terminal of the driver. Similarly the computer interface receives serial data from a TTY or CRT terminal through a line receiver. The line receiver converts 12 V (20 mA) and line open conditions into TTL 1 and 0 respectively. A receiver circuit is shown in Fig. 13.5 (e).

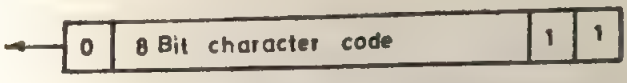
The printing mechanisms of teleprinters normally use impact printing method. In this method of printing, the characters (letters, digits and other symbols) are engraved on a small cylinder. When a character code is received by the TTY, the cylinder is positioned such that the character to be printed faces the ribon and it is hammered by the hammer activated by the mechanism. This thus prints the character on the paper.

CRT terminals are increasingly being used in place of teleprinters. This trend was set by the large scale integrated circuit technology which made the electronic circuits used in CRT displays inexpensive. A CRT display uses a ROM character generator which stores the dot patterns for the complete character set. For example, the letter E may be stored as a $7 \times 5$ array of 1's and 0's (1 for a dot and 0 for no dot) as shown in Fig. 13.5 (f). These 1's and 0's are read in sequence and CRT beam is intensified if 1's are read to produce dots on the CRT screen. A CRT controller (available commercially as a LSI chip) controls the retrieval of dots and


(a)

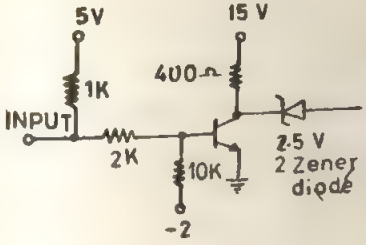



(b)

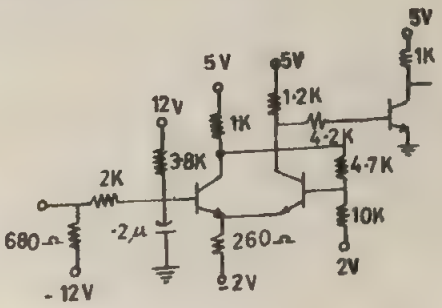



(c) 11 Bit shift register to transmit the data to TTY

(c)




(d)




(e)

(f)

FIG. 13·5

beam movement along X and Y directions on the screen, such that the given character is displayed at the required position.

Some CRT terminals may have capabilities to store and manipulate (edit) the page of text which was keyed in the terminal. Such terminals are called intelligent terminals, since they have certain computing power. With the advent of microprocessors (see Chapter 14) CRT terminals are becoming progressively more intelligent.

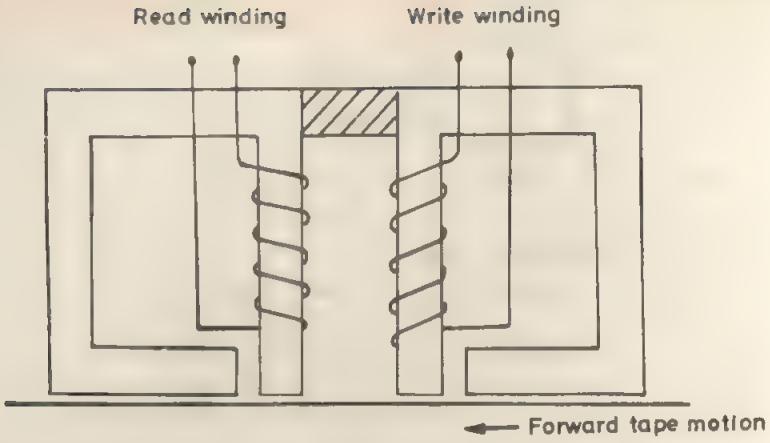



FIG. 13·6 MAGNETIC HEAD SCHEMATIC

## 13.6. MAGNETIC TAPE AND DISK DEVICES

Magnetic tape and disk devices are commonly used for large scale data storage. The storage of 100 million bytes of data is usually common using a single magnetic tape or disk pack. These devices use surfaces coated with the magnetic material as recording media. Data is written on the moving medium using the magnetic Read/Write head (Fig. 13.6) by passing the appropriate current pulses through the Write winding of the head. The recorded data is read using the Read winding. This is done as follows. When the medium (say tape/disk surface) moves with respect to the head, variations in the magnetic flux cause the voltage to be induced in the read winding. This voltage is amplified and shaped to produce logic signal levels.

### 13.6.1. Recording Techniques

The recording of 1's and 0's on a tape or disk track may be done in a number of ways. In this section, we discuss various methods.

**RZ (Return to Zero) Recording**

Fig. 13.7 (*a*) illustrates the RZ recording

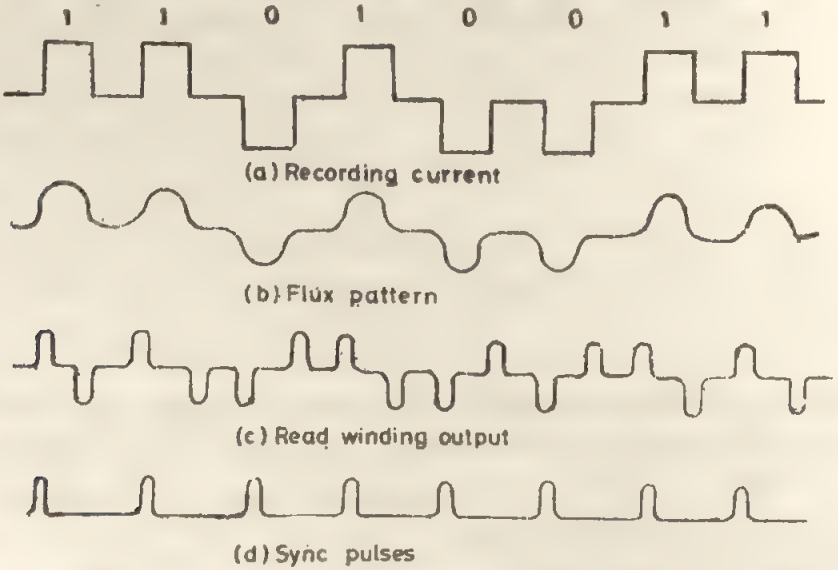



FIG.13·7. RZ RECORDING

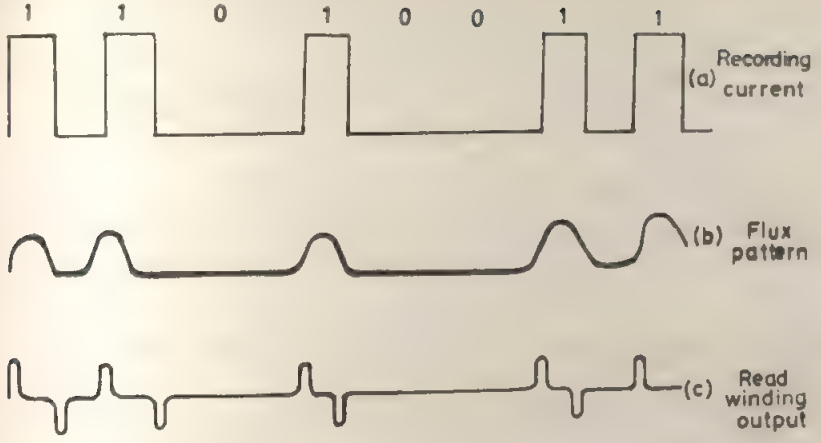



FIG. 13·8 RB RECORDING

waveform for the data pattern shown. In this recording method, a 1 is recorded with positive current pulse, while 0 is recorded with a negative current pulse. In either case, the current is returned to zero value after the pulse (hence the name RZ). Fig. 13.7 (*b*) illustrates the recorded flux on the track. The track with this flux pattern, when passes under the head, the voltage waveform shown in Fig. 13.7 (*c*) is induced in the read winding of the head. The sync pulses (timing pulses) generated by the timimg track (or any other means) are shown in Fig. 13.7 (*d*). The output voltage from read winding is amplified and shap.1 into the logical signal voltage levels. The serial data bits thus read are assembled in a register and transferred into the main memory of the computer.

**Return to Bias Recording**

In this technique, a 1 is recorded with a positive current pulse (+ I) and the current returns to a negative value (—I) and remains negative till an another 1 is encountered in the bit stream of the data. In other words, 0's are recorded with the biase value (— I) of the current. Fig. 13.8 (*a*) shows the write head current pulses for the data shown and the magnetisation is shown in Fig. 13.8 (*b*). The read winding output is shown in Fig. 13.8 (*c*).

**NRZ (Non Return to Zero) Recording**

Fig. 13.9 illustrates the NRZ recording

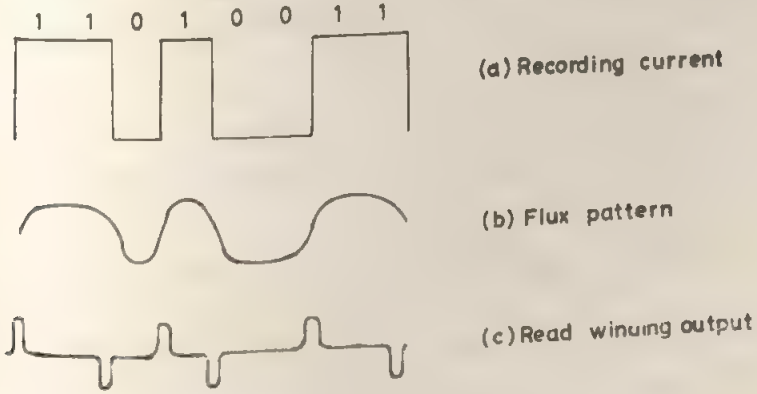



FIG.13·9 NRZ RECORDING

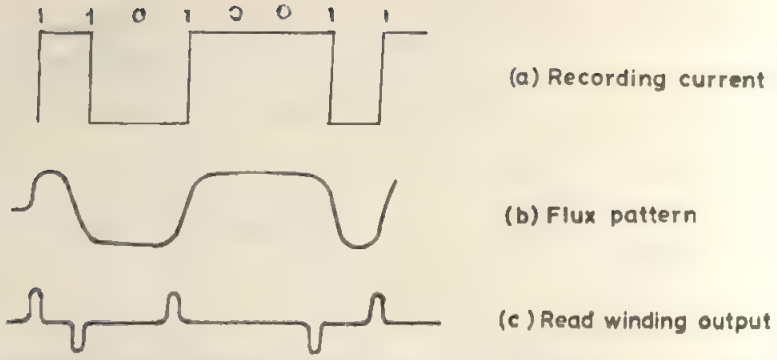



FIG. 13·10 NRZI RECORDING

waveforms. In this method of recording, a 0 is recorded with the negative current for the entire bit time, and 1 is recorded with the positive current for the entire bit time.

**NRZI Recording**

A variation of NRZ recording is possible. In this variation the recording current is reversed in the polarity every time a 1 is encountered in the bit stream to be recorded, otherwise the current remains constant. This method is known by the name NRZI (Non Return to Zero Inverted). Fig. 13.10 (*a*) shows the recording current waveform, while Fig. 13.10 (*b*) and Fig. 13.10 (*c*) show the flux pattern and the read winding voltage respectively.

**Phase Encoded Recording**

Phase encoded recording also called Ferranti or Manchester recording is an yet another variation of NRZ recording. In this recording technique, a 1 is recorded by a positive current for a half of the bit time followed by a negative current for the remaining half of the bit time. A 0 is recorded with a negative current for half bit time followed by the half bit time positive current. Fig. 13.11 shows the waveforms for this recording method.

**Discussion of Various Recording Techniques**

We studied various recording techniques in the present section. It can be noted that in RZ recording the magnetic flux returns to zero value after recording each bit. This makes it impossible to use it practically. This is because, it is not possible to write over the previously recorded information, unless the position of each cell is accurately located. For example, suppose that on some position on the track, we recorded a bit (say with the value 1) using RZ method. This will have a positive flux pattern followed by no flux pattern (for zero part of the current).

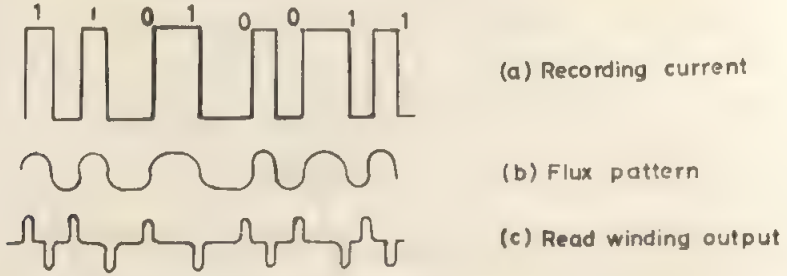
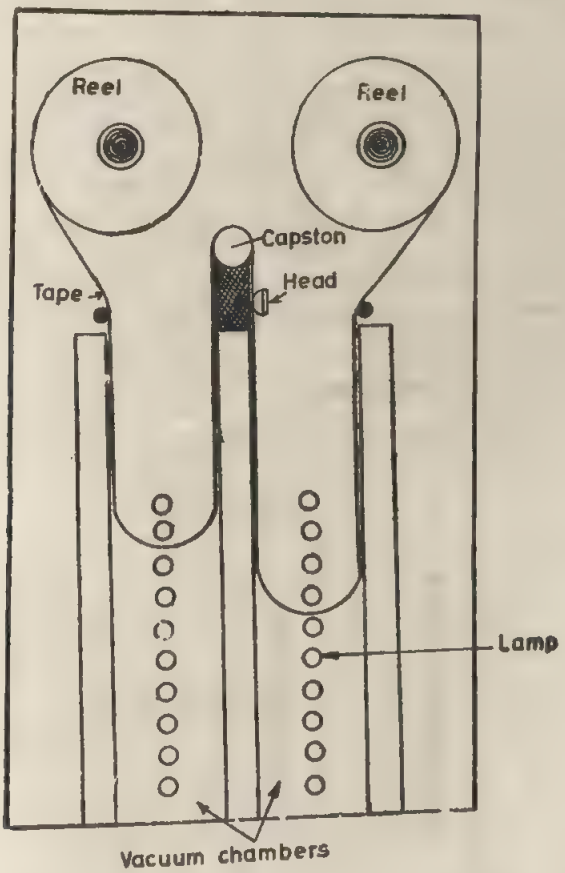



FIG. 13·11 PHASE ENCODED RECORDING

Now suppose we want to record on this track, and suppose cell positions (bit positions) are shifted slightly. It may happen that, zero value of a current in the new recording may fall at the position of track where a positive flux was already present (due to the previous recording). Since the zero current (in new recording) is not capable of changing the flux, the data will be erroneously recorded. Therefore the RZ method is rarely used. In the RB recording, a sequence of zeroes may cause problem, since 0's do not involve any change of flux. This method can be used if a special track for clock is used or, at least one track has a 1 on every cell position. Phase encoded recording has the advantage of having flux changes for every recorded bit. Same is not the case with NRZ or NRZI techniques. Phase encoded recording is now used on 1600 BPI (bits per inch) tape drives, while NRZI was used in 800 BPI tapes in earlier computers.

### 13.6.2. Magnetic Tape Devices

Magnetic tape provides an inexpensive and convenient medium for the storage of large amount of data. Also these tapes, unlike paper tapes can be used again and again,




Lamps indicate the depth to which the tape has gone in the vacuum chamber

FIG. 13·12 MAGNETIC TAPE DRIVE

since we can erase the old data and write new one in its place.

Magnetic tape system consists of a tape transport mechanism and a read/write station through which the magnetic tape passes. The transport mechanism consists of a motor driven capstan, with good breaking and take off arrangements, so that the tape can be stopped accurately. Usually for high speed tape operation, vaccum breaking is employed. The tape start and stop times may vary from 1 millisecond to 8 milliseconds. The read/write station consists of a magnetic recording head with two gaps (See Fig. 13.6). One gap is used for write operation in which the flux is established by the current in the write winding, while the other gap generates the emf on the read winding. With such an arrangement, data while being recorded could be read and checked against the recorded data. This is possible because, the tape first will move under the write-gap and then pass through the read-gap. When the data from the tape is read in the reversed direction (this facility is available in some tape systems), the roles of read and write windings are interchanged. Fig. 13.12 shows a magnetic tape drive.

### 13.6.3. MAGNETIC DISK DEVICES

Like magnetic tapes, disks are used for large scale data storage, but disks have the advantages of better data accessing method. In magnetic tapes data has to be read/recorded sequentially *i.e.*, tape drive has to skip the tape till a desired position (where data is to be read/recorded) comes under the head. In magnetic disks, the data access is partly sequential and partly direct hence these devices are known as direct access sequential devices (DASD).

A magnetic disk pack consists of a number of circular disks attached to a common spindle as shown in Fig. 13.13. Each disk has two surfaces coated with the magnetic material. The disk pack schematic shown in Fig. 13.13 has 12 surfaces out of which two (top most and bottom most) are unused for the obvious reasons. The disk spindle is driven by the electric motor. The rotational speeds vary with the manufacturers, but typically they are of the order of 3000 rpm.

A surface of a disk is divided into a number of concentric circular tracks as shown in Fig. 13.14. Tracks may be numbered in a convenient manner. Tracks on all the surfaces bearing the same number forms a cylinder which bears the same number (of the tracks) (See Fig 13.14). The data on the disk device is recorded cylinderwise to save on the otherwise necessary time consuming

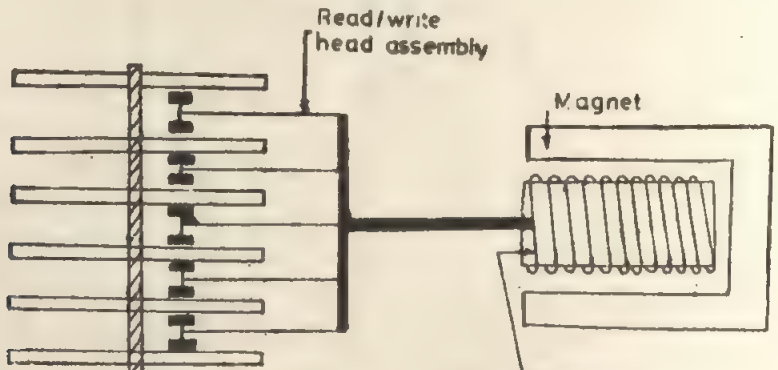



FIG. 13·13 MAGNETIC DISK DRIVE

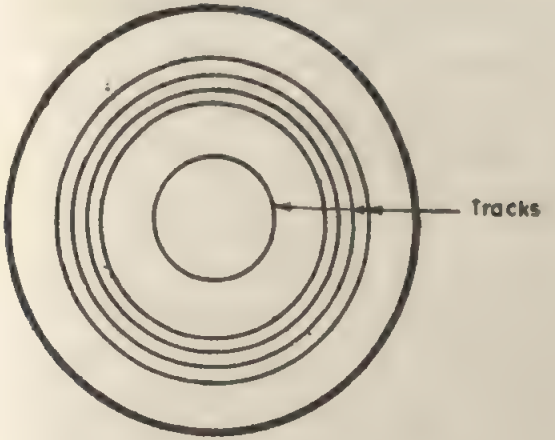



FIG 13.14 TRACKS ON DISK SURFACE

movements of the read/write heads.

A disk drive consists of a motor to rotate the disk assembly. The head assembly has to be moved in and out of the disk pack such that a required cylinder is selected for read/write operation. The mechanism for moving the head assembly consists of a solenoid which can linearly move in the field of a strong magnet. The head assembly is attached to the solenoid so that it moves whenever the solenoid is moved. Solenoid can be made to move in either of the directions by the current of appropriate polarity. The accurate positioning of heads is required to carry out reliable read or write operation. This calls for very accurate and precise positioning mechanism.

The time required to access data in the disk device consists of a seek time (time required to move the head assembly to the desired cylinder) and the latency time (which is required for data record to enter the read/write head position). The latency time thus, can at most be equal to the time for one revolution of the disks.

In costly disk systems, the seek time is nullified by providing one fixed head per track. In such systems heads are typically arranged in groups of eight or ten and are carefully aligned in fixed positions with respect to the disk surface. Since track densities greater than 60 per inch is difficult to achieve in fixed head disks, the storage capacity of a fixed head disk is less than that of a moving head disk with the same surface area.

## 13.7. I/O SYSTEMS

I/O system of a computer consists of hardware and software components through which I/O devices communicate to the computer system. I/O software subsystem is comprised of programs whose tasks involve initiation, optionally data conversion and overall supervision of I/O functions on various I/O devices. In the present section we are concerned with the hardware aspects of the I/O system. In digital computers I/O operations are carried out using one of the following methods.

* Programmed I/O
* Programmed I/O using interrupts
* I/O using channels or I/O processors.

### 13.7.1. Programmed I/O

In this method of I/O, CPU executes necessary instructions to carry out an I/O operation and is completely tied (busy) with the I/O operation. Since I/O devices are very slow compared to the CPU, it has to waste a lot of its time in waiting (idleing) for

the completion of the I/O operation, before an another operation could be started on the same device. Typically the programmed I/O involves the execution of instructions by CPU to carry out the following actions. These are separately described for input and output operations as follows :

**Input Operation**

(*i*) CPU starts the device for I/O operation and puts the device in the busy state (device flag is cleared).

(*ii*) CPU waits till the data is ready in the device buffer register. Device puts the data in the device buffer and indicates the data ready condition by setting the device flag.

(*iii*) Data from the device buffer is taken out by the CPU (either in a CPU register or main memory).

**Output Operation**

(*i*) CPU checks for the device flag (device status) and if flag is OFF, it waits for the flag to become ON. [*i.e.*, it repeats (*i*)].

(*ii*) CPU outputs the data into the device buffer register.

(*iii*) CPU starts the device for I/O operation.

It may be noted that the above actions were already introduced in Chapter 12, while studying the HM630 instruction cycles. In the HM630 I/O instruction cycle, the wait condition was implemented in hardware (by stopping the CPU clock and restarting it when the device flag becomes ON). The wait can also be implemented in software if, CPU has instructions to check the flags of the I/O devices. As an example of this we present the examples of input and output operations between CPU and paper tape devices in HP2100A minicomputer. The programs (written as subroutines) are as follows :

**Input Program**

| Label | Operation | Operands |
|---|---|---|
| INCH | NOP | |
| | STC | 9, C |
| WAIT | SFS | 9 |
| | JMP | WAIT |
| | LIA | 9 |
| | JMP | INCH, I |

**Output Program**

| Label | Operation | Operands |
|---|---|---|
| OUTCH | NOP | |
| WAITO | SFS | 10 |
| | JMP | WAITO |
| | OTA | 10 |
| | STC | 10, C |
| | JMP | OUTCH, I |

These programs have been written so that they could be called as subroutines. The input routine when called with the JSB INCH instruction stores the return address in the location assigned to the label INCH (NOP instruction is replaced by the return address). Thus, when the instruction JMP INCH, I is executed at the end of routine control will go back to the main program. Similar actions take place in the output routine. The instruction STC 9, C in the input routine starts the device number 9 (paper tape reader) for the I/O operation and clears its flag (C, option in the instruction STC 9, C). The instruction SFS 9 (skip if flag of device 9 is ON) checks the flag of device 9 and skips the next instruction if it is ON, otherwise the instruction JMP WAIT is executed and thus, checking (waiting) for the flag is repeated. When the flag becomes 1 (ON), control comes to the instruction LIA 9, which loads the data byte from device buffer register into the A register in the CPU.

The output routine works similarly. Here data from CPU (A register) is outputed to the device buffer of the tenth device (paper tape punch), and if the device flag is ON

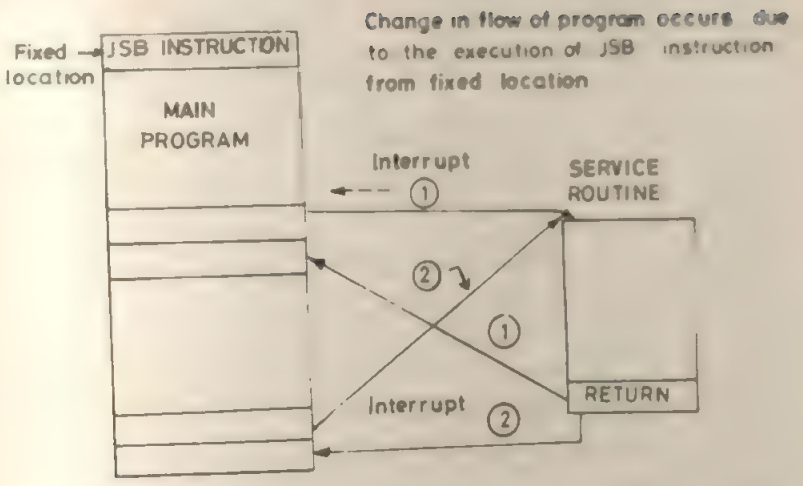



FIG. 13·15 SCHEMATIC SHOWS INTERRUPTS AND CHANGE IN FLOW OF PROGRAM

(device free condition) CPU issues the STC 10, C instruction, which starts the device for I/O operation and at the same time clears the device flag). After this the instruction JMP OUTCH I, (jump indirect) transfers the control back to the calling program.

### 13.7.2. I/O Operations Using Interrupts

In the programmed I/O discussed in the earlier subsection, it can be noted that the CPU is completely tied down during I/O operation. It is possible to relieve the CPU from this unnecessary waste of time by making use of interrupt facilities. The concept of interrupt was invented mainly due to the need for better CPU utilisation while carrying out the I/O operations.

An interrupt facility (if available) can be used to execute the subroutine (I/O routine) through the link from a fixed location. In this, whenever an interrupt condition (device flag ON condition in case of I/O interrupts) occurs, an instruction (usually a jump to subroutine instruction) from a fixed location is executed. This instruction passes the program control to the I/O routine (called I/O service routine).

The I/O routines discussed earlier for programmed I/O, must be modified as follows. The major change will be the removal of the instructions which were repeatedly executed making CPU to wait. Also the another important change required is the addition of instructions to save and restore the status of the interrupted program (contents of various registers, the ALU conditions, etc.). The saving of the status is done just after the entry to the service routine, while the restoring is done prior to the return instruction. The execution of service routine is automatically (Fig. 13.15) done whenever the device flag becomes ON (device end condition). Thus an I/O operation initiated earlier, when gets completed, will set the device flag and generate an interrupt condition. The interrupt condition makes CPU to execute an instruction from a fixed location. This instruction (usually a jump to subroutine) causes the control to go to the I/O service routine. This process can continue indefinitely, but CPU has instructions which may under certain conditions disable (mask off) the interrupt. For example, suppose a program needs 1000 bytes of data from

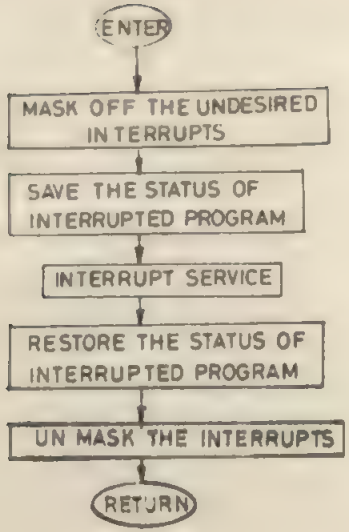



FIG. 13·16 INTERRUPT SERVICE ROUTINE SCHEMATIC

paper tape reader. What is needed is the execution of I/O service routine 1000 times. This can be achieved by initiallising a counter to 1000 in the main program and decrementing and checking the zero of the counter in the service routine. If counter reaches zero the interrupt disable instruction is executed after which, the CPU does not recognise the interrupt and thus no more interrupts (no more execution of I/O service routine) can occur. Fig. 13.16 shows a flow-chart which summarises the actions involved in writing an interrupt service routine.

**Hardware Design for Interrupts**

Here we shall discuss the hardware aspects of implementation of the interrupts in computers. For simplicity let us assume that we have only one source of interrupt. The hardware logic required to provide the interrupt facility is typically as follows :

An interrupt request is registered by setting a flip-flop to 1 (device flag in case of I/O interrupts). The CPU interrogates the state of this flip-flop at the end of every instruction and if it finds the flag ON, the internal (to CPU) IRQ flip-flop is set. The instruction fetch cycle (fetch microprogram in case of microprogrammed CPU) is designed, such that if the IRQ flip-flop is ON, then the instruction from fixed location is fetched (without affecting the program counter), otherwise the instruction pointed by the program counter is fetched. The fixed location usually stores the jump to subroutine instruction pointing to the interrupt service routine.

Multiple interrupts are handled as follows. Each source of an interrupt has a flip-flop for registering the interrupt request. Also each interrupt is assigned a fixed memory location in the main memory, where the jump to subroutine instruction is stored for linking the interrupt service routine with the interrupt. Since it is possible to have more than one interrupt flip-flops ON at a time, CPU must know the priority in which the interrupts are to be serviced. This is decided by a priority logic made of a multi-output combinational logic network, which maps the combinations of the interrupt flags (flip-flops) into the desired fixed addresses. Also, in case of multiple interrupts CPU will have instructions to enable or disable interrupts

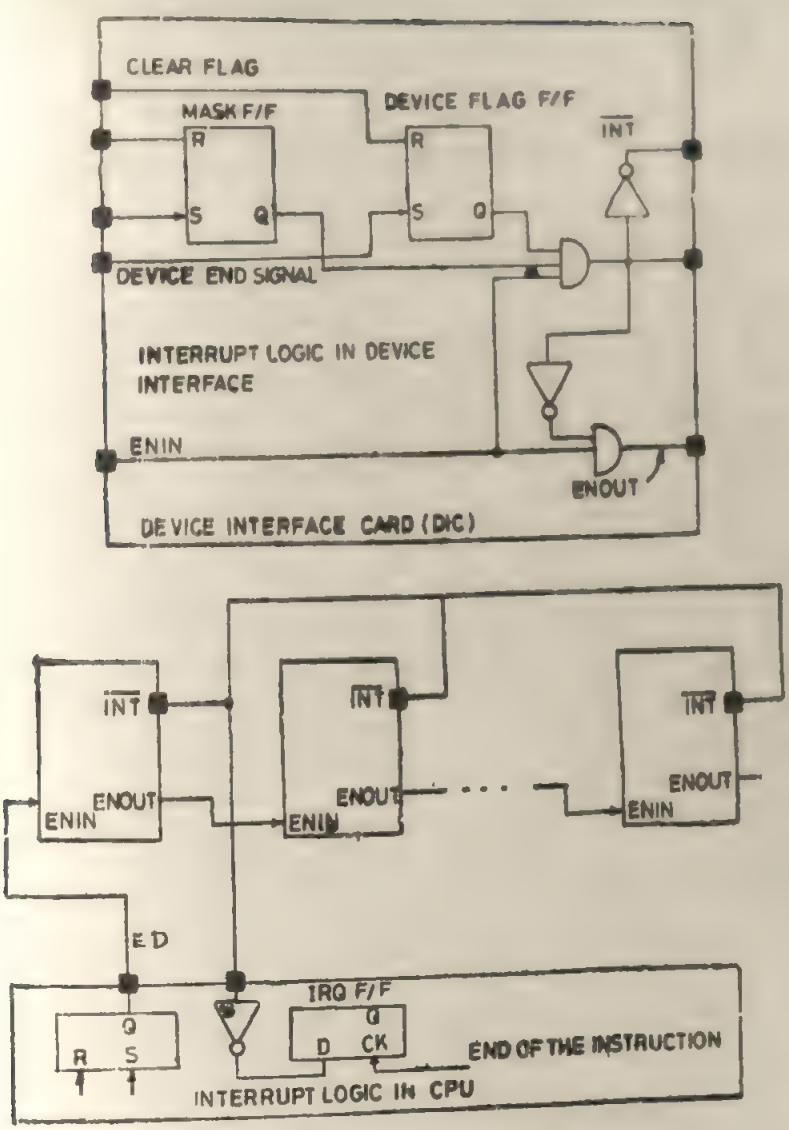



FIG 13.17 INTERRUPT HARDWARE

selectively. This is done by having one additional Mask flip-flop associated with each interrupt. In this arrangement an interrupt can occur only when its Mask and flag flip-flops are 1.

A popular arrangement to decide on the priorities is the use of daisy chain discussed as follows. In Fig. 13.17, we have an interrupt Enable/Disable (ED) flip-flop which can be set or reset by the CPU. The Q output of this flip-flop is passed through the interface circuits (cards) of the I/O devices in a predetermined order in which the priorities are required. The closest card (electrically) to the ED F/F gets the highest priority. The ENIN (Enable In) signal (Fig. 13.17) entering a device interface card, if true, indicates that there are no devices to the left (in the chain of Fig. 13.17) needing the interrupt and hence current device can cause the interrupt, if it is not masked off (mask F|F=1). The interrupt

logic present in the interface card performs the following tasks :

(*a*) If the interrupt is not masked off (MASK F/F=1) and flag F/F is 1, then it passes the interrupt request to the CPU by raising the INT line (The $\overline{INT}$ lines of all the cards can be OR tied).

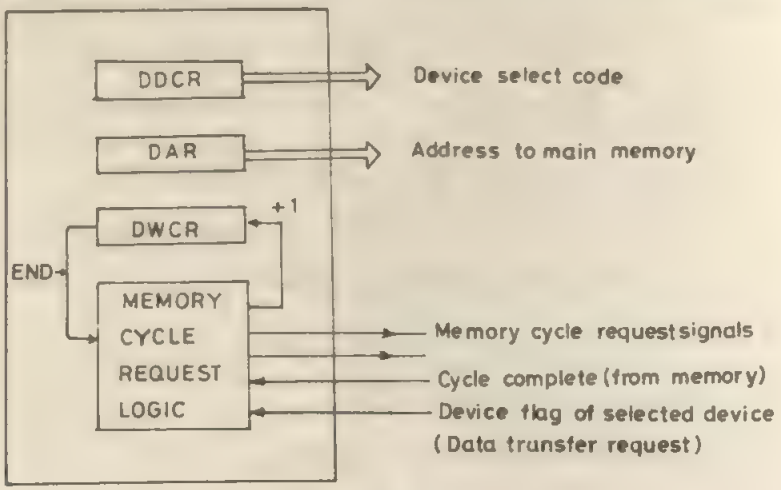
(*b*) Provides ENOUT line to be used as ENIN line by the next interface card in the daisy chain. ENOUT logic is shown in Fig. 13.17.

CPU recognises the interrupt by interrogating the $\overline{INT}$ line (a 0 on the $\overline{INT}$ line indicates the interrupt condition). The address of the interrupting device is put out on the bus by INT signal thus informing the CPU about the interrupting device. CPU may use this address to form the address of the fixed location (associated with interrupt) from where the interrupt instruction is to be executed. The service routine may disable all the further interrupts by making ED line 0 (by resetting ED flip-flop) or it may selectively reset the Masks of various interrupts.

### 13.7.3 Channels and I/O Processors

The third mode of I/O communication is the use of I/O channels, designed to carry out the data transfer (independent of CPU) between I/O devices and the main memory. Use of channels not only relieves the CPU almost completely from the I/O activity, but its use is inevitable in certain cases where the CPU cannot carry out I/O with some devices due to the high data transfer rates involved. (For example, in case of magnetic tape or disk devices).

The channels can be designed with varied complexities. Simplest of the channels are DMA channels. A DMA channel hardware consists of an address register (to indicate the main memory address), word count register (to indicate the number of words to be transferred) and a register for holding the address of the device for which data transfer is to be carried out. DMA channels are usually used to carry out a block transfer of data. CPU has instructions to initiallise the DMA channel registers to the required values. After the initiallisation CPU gives go ahead (start I/O) instruction to the DMA channel. The DMA channel looks at the flag (by hardware logic) of the selected device and if the flag is 1, initiates an access cycle to the main memory to carry out the data transfer.




DDCR DMA Device Code Register
DAR DMA Address Register
DWCR DMA Word Count Register

FIG. 13·18 DMA BLOCK DIAGRAM

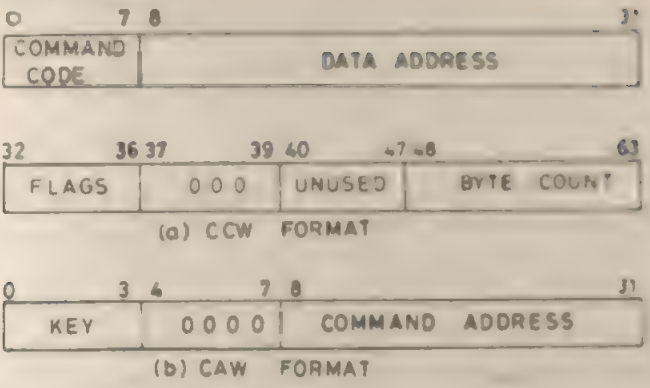



FIG. 13·19 CCW / CAW FORMATS

The memory access logic in the DMA channel generates memory cycles at the rate of device (by looking at the device flag). Since the CPU is also executing some program, the DMA channel and CPU may request memory access simultaneously. In such cases, priority is given to the channel and CPU simply waits (hardware wait). This way DMA steals one memory cycle from CPU (CPU would have otherwise got the memory cycle) and carries out the data transfer. The access method just discussed is called cycle stealing (since DMA channel steal memory cycles from CPU). In simpler designs, when DMA needs an access to the main memory, it may simply stop the CPU clock, irrespective of whether the CPU needs memory cycle or not, thus putting it in the wait state. The CPU is started when DMA cycle is complete. The block diagram of a DMA channel is shown in Fig. 13.18. Some machines have more sophisticated channels. The DMA channel discussed here is passive, while the I/O channels used in IBM 360/370 machine are highly sophisticated. These channels like CPU can execute their programs. The IBM 360/370 channels have double word (64 bits) commands (channel instructions are termed as commands). The command format of these channels is shown in Fig. 13.19 (a). Here the operation code part indicates the operation to be performed on the I/O device. Besides the usual read and write operations, a number of other operations could be coded in the channel command word (CCW) opcode. These will depend upon the device being serviced by the channel. For example, in case of magnetic tape devices, the opcodes could indicate rewinding of the tape, skipping the block of data etc. Similarly in the disk devices, they could code seek, select a given cylinder and so on.

To carry out an I/O operation, the IBM 360/370 CPU, just issues a SIO (Start I/O) instruction to the channel. This instruction specifies the device and channel numbers taking part in the I/O operation. Channel control logic finds out the status of the channel and if it is free, channel starts the I/O operation, otherwise the appropriate message code is returned to the CPU. The channel carries out an I/O operations by executing the channel program. The starting address of the channel program is indicated by the channel address word (CAW) prestored in the location 64. Fig. 13.19 (b) shows the CAW format.

The hardware and other details of these channels are beyond the scope of this book. Interested readers may refer to the bibliography given at the end of the book.

### I/O Processors

The channels discussed in the earlier discussion can independently carry out the I/O operations without much of the CPU's help (except in the beginning or end of the block transfer and when error conditions

occur). The use of an I/O processor for I/O operation is one step fu.ther in this direction. The I/O processor is by itself a processing unit like CPU and it is capable of carrying out data formating and editing functions, besides, it can completely look after the I/O operations including the handling of error conditions. The examples of a computer system using I/O processors is the famous large scale scientific machine CDC 7600 which have a number of I/O processors.

## 13.8 ERROR DETECTION AND CORRECTION

Error detection is the process of detecting signals which are different from those appearing in a properly operating system. Errors may occur due to permanent or intermittent faults in components of a system. In digital computers, errors in data can occur due to faults in digital circuits, corruption of data in storage media or malalignment of a medium with respect to the read/write head. Also dust particles on the medium may cause errors in the data.

The probability of errors occurring within CPU is much smaller than the probability of errors occurring in the main memory or I/O devices. It is a common practice in almost all computers to have error detection logic included in magnetic tape/disk device interfaces and main memory systems. Some machines also employ the error correction circuits for the main memories. In this section, we shall study the coding techniques for single error detection and correction.

### 13.8.1. Parity Checks

We discuss here the parity checking scheme which is most popular in its use. In parity checking, we associate additional digit or digits called as parity digit or digits with the information (message). A parity digit is chosen to satisfy certain criteria. In case of binary data, the value of a parity bit is chosen such that, the total number of 1's in the parity coded message is either even or odd. When the number of 1's is even, the checking scheme is called even parity check. On the other hand, if the number of 1's is odd, the checking scheme is called odd parity check.

**Example 13.1** : Using (*i*) even and (*ii*) odd parity checks find the parity bit for the message 11011011.

(*i*) Parity bit for even parity=0

(*ii*) Parity bit for odd parity=1

The coded message is :

| Message bit | Parity bit | |
|---|---|---|
| 1 1 0 1 1 0 1 1 | 0 | even parity |
| 1 1 0 1 1 0 1 1 | 1 | odd parity. |

### 13.8.2. Single Error Detection

Error occurring in a single bit can be easily detected using parity checks. The parity bit $p$ for even parity check can be expressed as :

$$p = a_0 \oplus a_1 \oplus a_2 \oplus \ldots a_{n-1} \qquad \ldots(13.1)$$

where $a_0, a_1 \ldots a_{n-1}$ are $n$ bits of the message and $\oplus$ denotes the XOR operation. Similarly the parity bit for odd parity scheme is given by

$$p = a_0 \oplus a_1 \oplus a_2 \oplus \ldots a_{n-1} \oplus 1 \qquad \ldots(13.2)$$

Error detection using parity checks is carried out as follows. At the transmitting end, the parity check bit for a data word (message) is calculated using one of the above parity check equations and the parity coded message (the data word with its parity bit) is transmitted. At the receiving end, the parity bit is calculated for the received message bits. This parity bit is compared with the incoming parity bit for the equality, and if they do not match, an error indication is given.

**Example 13.2** : Consider the even parity check on the data 10101101.

The parity bit $p$ is :

$p = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 1$.

Thus the parity coded word is 10101101 1 (where the right most bit is the parity bit).

Let the message received be 111011011. This message is erroneous with an error in

one bit. The parity bit $p_r$ is calculated at the receiver and is compared with the incoming parity bit.

Thus $p_r = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$

The $p_r$ value (*i.e.* 0) differs from the incoming parity bit (*i.e.* 1) and hence there exists an error in the received message.

The parity checking schemes discussed can detect any single bit error, or errors in the odd number of bits. This is possible because any change in a single bit or a change in the odd number of bits of the message will cause change in the parity calculated at the receiver (this happens due to the property of XOR operation). On the other hand, if there exists even number of errors, the parity bit value calculated at the receiver will be the same as that of the received parity bit, and thus no error indication will be given. Thus parity check (with single parity bit) cannot detect even number of errors. Parity circuits are easy to implement (implementation of equation 13.1 or 13.2) using XOR gates and their design is left to the readers.

### 13.8.3. Single Error Correcting Hamming Codes

Hamming was the pioneer in the field of error detection and correction in digital communication systems. Hamming formalised various ideas in coding and gave number of results on the error detecting and correcting capabilities of codes. In the present section, we informally discuss the single error correcting codes (for binary cases) invented by Hamming.

To bring out the concepts involved in single error correcting Hamming codes, we present the following 7-bit message to the readers :

| $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Let us make one error in a bit say $b_6$ of the message. The message received with this error, thus will be

| $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |

Now we define the three parity relations as follows and evaluate the parities $p_4$, $p_2$ and $p_1$ with these relations for the received message.

$$p_4 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 \quad \text{...(13.3)}$$

$$p_2 = b_3 \oplus b_6 \oplus b_6 \oplus b_7 \quad \text{...(13.4)}$$

$$p_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 \quad \text{...(13.5)}$$

Evaluating $p_4$, $p_2$ and $p_1$ for the erroneous message we have,

$$p_4 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$p_2 = 0 \oplus 1 \oplus 1 \oplus 1 = 1$$

$$p_1 = 1 \oplus 1 \oplus 1 \oplus 1 = 0$$

The binary number formed by $p_4 p_2 p_1$ is 110. This number is same as the position number of the erroneous bit *i.e.*, $b_6$. Thus, by calculating $p_4$, $p_2$ and $p_1$ we know the position of erroneous bit in the message, and the correct message can be obtained by complementing the erroneous bit. On the other hand, if the received message is same as the original message, $p_4 p_2 p_1$ will yield a number 000, indicating no error.

The message taken for the discussion was Hamming coded and therefore we could obtain the correct message. This message has 4 information bits in positions $b_3, b_5, b_6$ and $b_7$, while other three bits in the message are the parity bits The **coding** of the 4-bit information is done using the following parity relations, *i.e.*, three parity bits (in positions $b_1, b_2$ and $b_4$) are obtained using the following relations :

$$b_1 = b_3 \oplus b_5 \oplus b_7 \quad \text{...(13.6)}$$

$$b_2 = b_3 \oplus b_6 \oplus b_7 \quad \text{...(13.7)}$$

$$b_4 = b_5 \oplus b_6 \oplus b_7 \quad \text{...(13.8)}$$

The 4-bit message with appropriate bit positions is shown below :

| $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | x | 1 | x | x |

The positions marked x are to be filled. Using Equations 13.6, 13.7 and 13.8, we have:

$b_1 = 1 \oplus 1 \oplus 1 = 1$

$b_2 = 1 \oplus 0 \oplus 1 = 0$

$b_4 = 1 \oplus 0 \oplus 1 = 0$

The Hamming coded message thus is :

| $b_7$ | $b_6$ | $b_5$ | $b_4$ | $b_3$ | $b_2$ | $b_1$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |

The coded message has the single error correcting capability as illustrated earlier.

In summary, what we have done is as follows. For a given 4-bit message, we calculated the three parity bits using the Equations 13.6, 13.7 and 13.8 at the transmitting end. This gave us the 7-bit Hamming coded message. The coded message is received at the receiver (may be with an error). At the receiver, we decoded the message by calculating the parities $p_4$, $p_2$ and $p_1$ using relations given in Equations 13.3, 13.4 and 13.5. The number formed by $p_4\ p_2\ p_1$ gave us the position of erroneous bit and from this we could obtain the correct message by complementing the erroneous bit.

The error correcting capability of the Hamming code can be qualitatively explained as follows. We again consider the example message, where there was an error in the 6th bit. A look at the Equations 13.6, 13.7 and 13.8 tells us that they define even parities for the respective bit positions (bit positions occurring in the equations), *i.e.*, the bits in (1, 3, 5, 7), (2, 3, 6, 7) and (4, 5, 6, 7) should have the even parities in the transmitted message. At the receiver, we are just checking the violations of these parities using Equations 13.3, 13.4 and 13.5. The violation of parities is indicated by $p_4$, $p_2$ and $p_1$ in position groups (4, 5, 6, 7), (2, 3, 6, 7) and (1, 3, 5, 7) respectively.

In the earlier example message we had $p_4\ p_2\ p_1 = 110$. The information about error given by $p_4\ p_2\ p_1$ can be explained as follows. Any of the variables $p_4$, $p_2$ or $p_1$ taking a value 1 indicates the violation of a corresponding parity relation.

Thus,

$p_4 = 1$ implies an error in one of the bits $b_4$, $b_5$, $b_6$ or $b_7$

$p_2 = 1$ implies an error in one of the bits $b_2$, $b_3$, $b_6$ or $b_7$

$P_1 = 0$ implies no error in any of the bits $b_1$, $b_3$, $b_5$ or $b_7$.

Since there is no error in $b_1$, $b_3$, $b_5$, $b_7$ (because $p_1 = 0$), we can remove these bits from the lists of bits indicated by $p_4$ and $p_2$. This gives us :

$p_4 = 1$ implies error in $b_4$ or $b_6$

$p_2 = 1$ implies error in $b_2$ or $b_6$

Since there is single error (assumption) the error must be in a common bit indicated by $p_4$ and $p_2$. This gives us $b_6$ as the erroneous bit.

The above discussion concerned only with coding of the 4-bit information. In general to code an n-bit message for single error correction, we require k parity check bits. The number k is given by :

$$k = [\log_2 (n + k)]$$

where the notation [x] denotes the smallest integer $\geq x$. For example, if we have 10-bit message to be Hamming coded, we shall require :

$$k = [\log_2 (10 + k)]$$

*i.e.*, 4, check bits.

Next step in coding is to place these check bits in the positions corresponding to the powers of 2, *i.e.*, check bits shall occupy positions 1, 2, 4, 8, 16, etc., in the coded message. This placing of check bits allows us to have only one unknown per parity equation (equations 13 6, 13.7 and 13.8), thus facilitating the easy evaluation of parities. The parity equations themselves are defined from the table of combinations of check bits. Table 13.3 gives the combinations of check bits for 7-bit Hamming coded message.

Table 13.3

| $p_4$ | $p_2$ | $p_1$ | Positions |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

A 1 in a particular column and row indicates the bit position associated with the corresponding parity equations. Thus $p_4$, $p_2$, $p_1$ have 1 in their columns indicating bit groups (4, 5, 6, 7), (2, 3, 6, 7) and (1, 3, 5, 7) respectively. In a general case there may be less or more columns (check bits) and rows.

**Example 13.3** : Illustrate the Hamming coding for the 10 bit message 1011001110.

Since we have 10 bits in original message, we shall require 4 parity bits. The parity bits shall be placed in the positions 1, 2, 4 and 8. The numbering of bits is arbitrary, provided same conventions are used in the transmitting and receiving sides. Here we deliberately number the bits from left to right.

$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$

| x | x | 1 | x | 0 | 1 | 1 | x | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

The parity check equations for coding are derived from the combinations of 4 check bits as discussed earlier, thus

$b_1 = b_3 \oplus b_5 \oplus b_7 \oplus b_9 \oplus b_{11} \oplus b_{13}$

$b_2 = b_3 \oplus b_6 \oplus b_7 \oplus b_{10} \oplus b_{11} \oplus b_{14}$

$b_4 = b_5 \oplus b_6 \oplus b_7 \oplus b_{12} \oplus b_{13} \oplus b_{14}$

$b_8 = b_9 \oplus b_{10} \oplus b_{11} \oplus b_{12} \oplus b_{13} \oplus b_{14}$

Thus

$b_1 = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$

$b_2 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 = 0$

$b_4 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$

$b_8 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 1$

These four values are filled in the message in the respective positions (marked x) and we get the Hamming coded message, given at the bottom of the page where bits circled are the parity bits.

**Example 13.4** : In the Hamming coded message of Example 13.3, create an error in the position 11 and illustrate the error correction.

Due to the error in bit position 11 the received message shall be :

$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

The parity checks on this message gives :

$p_1 = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1$

$p_2 = 0 \oplus 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 1$

$p_4 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$

$p_8 = 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$

This gives us $p_8$ $p_4$ $p_2$ $p_1$ = 1011, *i.e.*, eleventh bit. The correct message is obtained by complementing the $11^{th}$ bit and hence the corrected message is

$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$

| (0) | (0) | 1 | (0) | 0 | 1 | 1 | (1) | 0 | 0 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

## EXERCISES

1. List the various I/O devices and give their purpose in the computer system. Give brief discussion about their functional characteristics.

2. Magnetic tape and disk devices operate at a very high speed. Will the CPU be able to carry out the data transfer by programmed I/O technique ? Discuss.

3. What are the advantages of using interrupts over programmed I/O ? Discuss the various aspects.

4. Tap and disk devices usually communicate through the I/0 channels. Why ?

5. Discuss the interfacing of a teleprinter to computer.

6. How will you interface a card reader ? How does the 80 byte data on a card shall be read ? Discuss the procedure to be followed and its hardware implications.

7. Discuss the advantages of NRZ technique over RZ and RB recording techniques.

8. The error detection and correction techniques discussed can be economically employed only when message lengths are small. Why ?

9. The magnetic tape and disk devices store millions of bits. These are read serially and the probability of error occurring during read operation is unacceptably high. Can you suggest the way out ? Read the biblography on error detecting and correcting codes.

10. Encode the 8-bit message 11010110 into a single error correcting Hamming code.

11. Find the erroneous bit in the following Hamming coded message

| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | ←bit numbers |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | ←message |

12. Encode the 32-bit message $4ABC_{16}$ into single error correcting Hamming code.

13. Develop a logic circuit to implement a single error correction logic for a memory having 32-bit word length.

14. Encode the 10-bit message 1011101110 into a single error correcting Hamming code.

15. Following Hamming coded message is received. Find whether the message is erroneous (assuming at most single error) or correct. If the message is incorrect give the correct message

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | bit positions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | message |

16. It is desired to check the adder in the ALU for single errors in the sum bits. Can you suggest the parity checking scheme for detecting single errors in the sum bits ?

17. Suggest how to modify the FI (Fetch Instruction) cycle discussed in Chapter 1 ', to incorporate the interrupt feature in HM 630.

18. Repeat (17) for FETCH microroutine.

19. Tick all the appropriate statements (Note : more than one statement may be true).

(*i*) machine language instruction may require channels and I/O devices for their execution.

(*ii*) never require channel as I/O device for their executions.

(*iii*) Require ALU for their execution.

(*iv*) never use ALU.

20. Recording data cylinder-wise in magnetic disks causes :

(*i*) higher bit densities

(*ii*) higher data transfer rates

(*iii*) reduces the arm (head assembly) movements

(*iv*) none of the above.

□

# 14

# MICROPROCESSORS

## 14.1. INTRODUCTION

This chapter discusses the latest development in the computer technology, *i.e.*, microprocessors. The rapid growth of semiconductor technology has made possible to build a complete processor on a single IC chip Such processors are called microprocessors. The prefix micro was attached to the processor in mid 70's due to its size and capability, but presently, what we have are microprocessors equalling or exceeding the computing and architectural facilities available on a conventional minicomputer, with the trend that, they will soon reach the computing power of a CPU of large scale computer systems.

In the present chapter, we shall discuss the most popular of the microprocessor families *i.e.*, Intel Corporations 8080, 8085 and 8086 microprocessors and some of their support components.

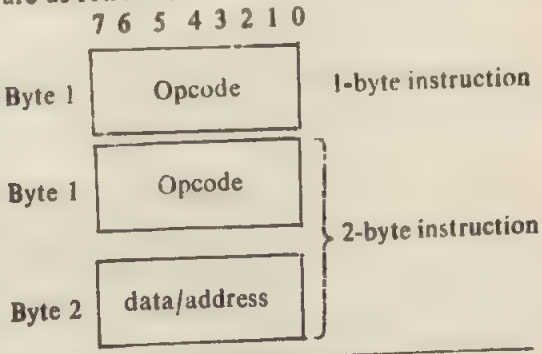
## 14.2. 8080 MICROPROCESSOR AND ITS SUPPORT COMPONENTS

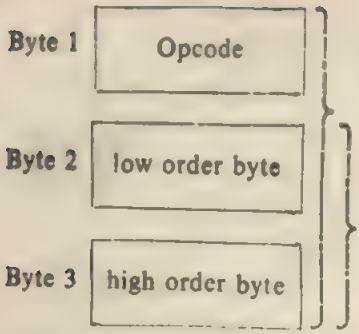
The 8080 microprocessor was introduced by Intel Corporation in 1973. It is fabricated on a single LSI chip and has 72 machine language instructions. The 8080 processor is packaged in a 40-pin DIP (dual in line) package and allows easy interfacing of other circuit elements. 8080 chip has 16-bit address bus and a separate 8-bit data bus and various control pins. It can address upto 64K bytes (1 byte=8 bits) of memory and 256 input and 256 output devices. It has seven, 8-bit general purpose registers, six of them could be used in pairs to from 16-bit registers.

### 14.2.1. Architecture

The main memory of 8080 processor is organised as 8-bit locations (bytes). Each byte has a unique 16-bit address. The 8080 can directly access upto 64K (1 K=1024) bytes of memory which may consists of ROMs and RAMs.

The 8080 has instructions with one, two or three bytes in length. A multiple byte instruction resides in consecutive memory locations. The various instruction formats are as follows :






The 8080 processor has 7 registers which are listed below. They are referred in the instructions by 3-bit codes (part of the opcode) given below :

| *Register* | *Length* | *3-bit code* |
|---|---|---|
| A | 8 bits | 111 |
| B | 8 bits | 000 |
| C | 8 bits | 001 |
| D | 8 bits | 010 |
| E | 8 bits | 011 |
| H | 8 bits | 100 |
| L | 8 bits | 101 |

Also, the six of three registers (excluding A register) can be addressed as pairs to refer the 16-bit data. These register pairs (rp) are addressed by 2-bit code given below :

| *Register Pair* | *2-bit code* |
|---|---|
| BC | 00 |
| DE | 01 |
| HL | 10 |
| SP (stack pointer) | 11 |

There are 5 condition flags in 8080 and these are listed and described as follows :

| *Flag* | *Description* |
|---|---|
| Z | Zero : This flag is set if the result of an operation is zero, otherwise it is reset. |
| S | Sign : Sign of the result |
| AC | Auxiliary: Carry from bit number 3. |
| P | Parity : It is set when there exists even number of 1s in the result, otherwise it is reset. |

| *Flag* | *Description* |
|---|---|
| CY | Carry : This flag is set as a result of a carry from the most significant bit. |

These flags are affected after the execution of certain instructions, while some instructions do not affect the flags.

## ADDRESSING MODES

8080 has the following four addressing modes.

Direct addressing

Register addressing

Register indirect

Immediate addressing.

In ***direct addressing***, the 16-bit address is specified in byte 2 and byte 3 of the instruction word. Byte 3 holds the higher order part and byte 2 holds the lower address part of the operand address.

In ***register addressing*** mode, the address of the register or register pair is specified by the 3-bit or 2-bit code respectively.

**Register indirect addressing** allows addressing of the operand in main memory location whose address is specified in a register pair.

In the ***immediate addressing***, the 8 or 16-bit data is available in the instruction word.

### Instruction Set

The list of the machine language instructions with their brief description is given in Table 14.1. More details, if required can be found from the manufacturer's reference manual.

TABLE 14.1

**8080/8085 Instruction Set**

| *Mnemonic* | *Description* | *Instruction Code* $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | *Clock Cycles* |
|---|---|---|---|---|---|---|---|---|---|---|
| | **MOVE, LOAD, AND STORE** | | | | | | | | | |
| MOVE *r*1, *r*2 | Move register to register | 0 | 1 | D | D | D | S | S | S | 5 |
| MOV M, *r* | Move register to memory | 0 | 1 | 1 | 1 | 0 | S | S | S | 7 |
| MOV *r*, M | Move memory to register | 0 | 1 | D | D | D | 1 | 1 | 0 | 7 |
| MVI *r* | Move immediate register | 0 | 0 | D | D | D | 1 | 1 | 0 | 7 |
| MVI M | Move immediate memory | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 10 |
| LXI B | Load immediate register Pair B and C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10 |
| LXI D | Load immediate register Pair D and E | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 10 |
| LXI H | Load immediate register Pair H and L | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 10 |
| STAX B | Store A indirect | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 7 |
| STAX D | Store A indirect | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 7 |
| LDAX B | Load A indirect | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 7 |
| LDAX D | Load A indirect | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 7 |
| STA | Store A direct | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 13 |
| LDA | Load A direct | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 13 |
| SHLD | Store H and L direct | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 16 |
| LHLD | Load H and L direct | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 16 |
| XCHG | Exchange D and E, H and L registers | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 4 |
| | **STACK OPERATIONS** | | | | | | | | | |
| PUSH B | Push register Pair B and C on stack | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 11 |
| PUSH D | Push register pair D and E on stack | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 11 |
| PUSH H | Push register pair H and L on stack | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 11 |
| PUSH PSW | Push A and Flags on stack | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 11 |
| POP B | Pop register pair B and C off stack | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 10 |
| POP D | Pop register pair D and E off stack | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 10 |
| POP H | Pop register pair H and L off stack | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 10 |
| POP PSW | Pop A and Flags off stack | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 10 |
| XTHL | Exchange top of stack H and L | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 18 |

| Mnemonic | Description | Instruction Code $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | Clock Cycles |
|---|---|---|---|---|---|---|---|---|---|---|
| SPHL | H and L to stack pointer | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 5 |
| LXI SP | Load immediate stack pointer | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 10 |
| INX SP | Increment stack pointer | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 5 |
| DCX SP | Decrement stack pointer | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 5 |
| | **JUMP** | | | | | | | | | |
| JMP | Jump unconditional | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 10 |
| JC | Jump on carry | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 10 |
| JNC | Jump on no carry | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 10 |
| JZ | Jump on zero | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 10 |
| JNZ | Jump on no zero | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 10 |
| JP | Jump on positive | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 10 |
| JM | Jump on minus | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 10 |
| JPE | Jump on parity even | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 10 |
| JPO | Jump on parity odd | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 10 |
| PCHL | H and L to program counter | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 5 |
| | **CALL** | | | | | | | | | |
| CALL | Call unconditional | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 17 |
| CC | Call on carry | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 11/17 |
| CNC | Call on no carry | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 11/17 |
| CZ | Call on zero | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 11/17 |
| CNZ | Call on no zero | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 11/17 |
| CP | Call on positive | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 11/17 |
| CM | Call on minus | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 11/17 |
| CPE | Call on parity even | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 11/17 |
| CPO | Call on parity odd | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 11/17 |
| | **RETURN** | | | | | | | | | |
| RET | Return | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 10 |
| RC | Return on carry | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 5/11 |
| RNC | Return on no carry | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 5/11 |
| RZ | Return on zero | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 5/11 |
| RNZ | Return on no zero | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 5/11 |
| RP | Return on positive | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 5/11 |
| RM | Return on minus | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 5/11 |
| RPE | Return on parity even | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 5/11 |
| RPO | Return on parity odd | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 5/11 |
| | **RESSTART** | | | | | | | | | |
| RST | Restart | 1 | 1 | A | A | A | 1 | 1 | 1 | 11 |
| | **INCREMENT AND DECREMENT** | | | | | | | | | |
| INR *r* | Increment register | 0 | 0 | D | D | D | 1 | 0 | 0 | 5 |
| DCR *r* | Decrement register | 0 | 0 | D | D | D | 1 | 0 | 1 | 5 |

| Mnemonic | Description | Instruction Code $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | Clock Cycles |
|---|---|---|---|---|---|---|---|---|---|---|
| INR M | Increment memory | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 10 |
| DCR M | Decrement memory | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 10 |
| INX B | Increment B and C registers | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 5 |
| INX D | Increment D and E registers | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 5 |
| INX H | Increment H and L registers | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 5 |
| DCX B | Decrement B and C | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 5 |
| DCX D | Decrement D and E | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 5 |
| DCX H | Decrement H and L | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 5 |
| | **ADD** | | | | | | | | | |
| ADD *r* | Add register to A | 1 | 0 | 0 | 0 | 0 | S | S | S | 4 |
| ADC *r* | Add register to A with carry | 1 | 0 | 0 | 0 | 1 | S | S | S | 4 |
| ADD M | Add memory to A | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 7 |
| ADC M | Add memory to A with carry | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 7 |
| ADI | Add immediate to A | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 7 |
| ACI | Add immediate to A with carry | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 7 |
| DAD B | Add B and C to H and L | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 10 |
| DAD D | Add D and E to H and L | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 10 |
| DAD H | Add H and L to H and L | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 10 |
| DAD SP | Add stack pointer to H and L | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 10 |
| | **SUBTRACT** | | | | | | | | | |
| SUB *r* | Subtract register from A | 1 | 0 | 0 | 1 | 0 | S | S | S | 4 |
| SBB *r* | Subtract register from A with borrow | 1 | 0 | 0 | 1 | 1 | S | S | S | 4 |
| SUB M | Subtract memory from A | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 7 |
| SBB M | Subtract memory from A with borrow | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 7 |
| SUI | Subtract immediate from A | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 7 |
| SBI | Subtract immediate from A with borrow | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 7 |
| | **LOGICAL** | | | | | | | | | |
| ANA *r* | Add register with A | 1 | 0 | 1 | 0 | 0 | S | S | S | 4 |
| XRA *r* | Exclusive or register with A | 1 | 0 | 1 | 0 | 1 | S | S | S | 4 |
| ORA *r* | Or register with A | 1 | 0 | 1 | 1 | 0 | S | S | S | 4 |
| CMP *r* | Compare register with A | 1 | 0 | 1 | 1 | 1 | S | S | S | 4 |
| ANA M | Add memory with A | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 7 |
| XRA M | Exclusive or memory with A | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 7 |
| ORA M | Or memory with A | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 7 |
| CMP M | Compare memory with A | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 7 |

| Mnemonic | Description | Instruction Code $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | Clock Cycles |
|---|---|---|---|---|---|---|---|---|---|---|
| ANI | And immediate with A | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 7 |
| XRI | Exclusive or immediate with A | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 7 |
| ORI | Or immediate with A | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 7 |
| CPI | Compare immediate with A | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 7 |
| | **ROTATE** | | | | | | | | | |
| RLC | Rotate A left | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 4 |
| RRC | Rotate A right | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 4 |
| RAL | Rotate A left through carry | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 4 |
| RAR | Rotate A right through carry | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 4 |
| | **SPECIALS** | | | | | | | | | |
| CMA | Complement A | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 4 |
| STC | Set carry | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 4 |
| CMC | Complement carry | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
| DAA | Decimal adjust A | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 4 |
| | **INPUT/OUTPUT** | | | | | | | | | |
| IN | Input | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 10 |
| OUT | Output | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 10 |
| | **CONTROL** | | | | | | | | | |
| EI | Enable Interrupts | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 4 |
| DI | Disable Interrupt | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 4 |
| NOP | No-operation | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |
| HLT | Halt | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 7 |



Note : SSS or DDD gives 3-bit source or destination register codes respectively.
M stands for location whose address is in HL Pair.

### 14.2.3. Hardware Architecture

This section discusses the hardware design of a microcomputer using the 8080 processor and its family components. Before the discussion of the design of a microcomputer, we present the basic LSI circuit chips which are required to form the microcomputer.

### 8080 PROCESSOR CHIP

The Intel 8080 processor chip is fabricated using *n*-channel MOS technology. The functional block diagram of the 8080 processor and the pin definitions of the chip are shown in Fig. 14.1.

#### Data and Address Bus

These are data and addressbus lines with tri-state capability, and they provide the bi-directional 8-bit data communication between the processor chip and the outside devices (memory or I/O devices as per the address on address bus) Also, during the first clock of every machine cycle, the 8080 processor outputs a status word on the data bus. The status word gives the information about the current machine cycle.

#### SYNC

This signal indicates the beginning of a machine cycle and it can be used to clock the status word (available on data bus) into the external register.

#### DBIN

The DBIN (Data Bus In) signal indicates to the external circuits that the data bus is in the input mode. This signal should be used to enable the gating of the data on to the 8080 data bus from memory or I/O devices.

#### $\overline{WR}$

The $\overline{WR}$ signal indicates that the data bus contains the data (from the 8080 processor)

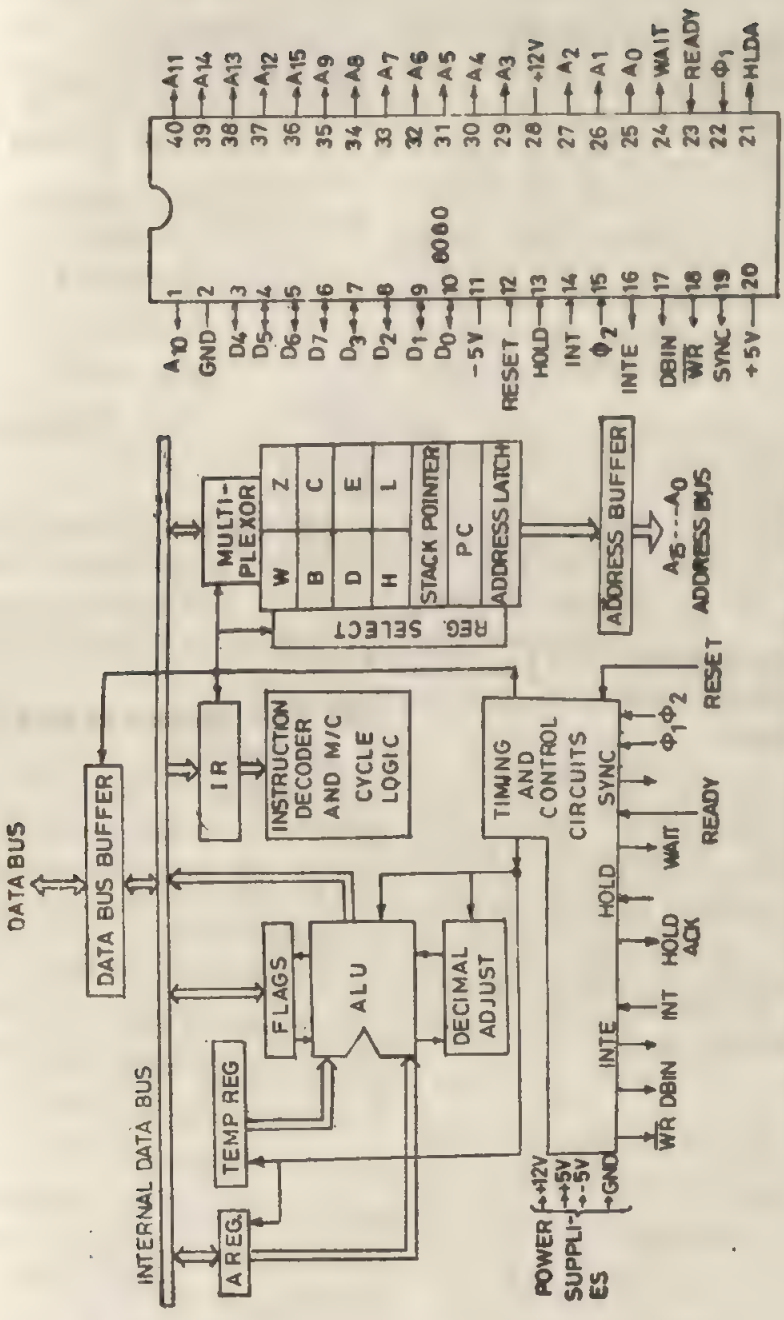



FIG 14-1 8080 MICROPROCESSOR HARDWARE ORGANISATION

which could be transferred to I/O or memory devices. This is active low signal (bar on WR). In other words, data bus contains the data till this pin is at the logical 0 level.

**READY and WAIT**

READY signal if made 0, puts the 8080 into the WAIT state (8080 temporarily suspends its activity). The processor remains in the wait state till the ready signal goes high (logical 1). The wait state of the processor is acknowledged by 8080 by producing a active high signal (logical 1) on the WAIT pin.

READY signal could be used to sychronise the 8080 processor with slow speed devices. For example, if 8080 wants a data from a memory device which requires more time than what 8080 normally allows, the 8080 could be put into wait state by making READY line low. The device puts the data on the data bus and raises the READY line to 1 state, and 8080 resumes its operation.

**HOLD and HLDA**

Hold signal requests the 8080 to enter into the HOLD state. Hold state of 8080 allows an external device (like DMA) to gain control of address and data buses as soon as the 8080 has completed the use of the buses for the current machine cycle. It puts the buses in the high impedance state. The 8080 acknowledges this by giving out a 1 on the HLDA (Hold Acknowledge) pin.

**INT and INTE**

The 8080 recognises an interrupt on the INT line at the end of the current instruction or while in HALT state. The interrupt is honoured only if it is enabled. The interrupt can be enabled by executing the EI (Enable Interrupt) instruction. The true output of the Enable flip-flop is available on INTE line.

**RESET**

The reset signal clears the program counter and initialises the 8080 processor. This makes INTE, HLDA lines and program counter 0 and the program execution starts from location 0 as soon as RESET line is made inactive. Note that none of the other registers are cleared by RESET signal.

**$V_{SS}$, $V_{DD}$, $V_{CC}$ and $V_{BB}$**

These are power supply lines. The required power supplies are (within $\pm 5\%$ tolerance)

$V_{DD} = +12$ V

$V_{CC} = +5$ V

$V_{BB} = -5$ V

$V_{SS}$ = Ground (reference)

**$\phi_1$ and $\phi_2$**

These are the clock pulses to be externally supplied for the operation of 8080 chip.

All the pins of 8080 processor chip are TTL compatible, except the two clock pulses. $\phi_1$ and $\phi_2$.

**8224 Clock Generator for 8080**

The 8224 provides clock pulses $\phi_1$ and $\phi_2$ to the 8080 CPU. Besides, it also provides buffering (more driving capacity) for some signals. The block diagram of 8224 is shown in Fig. 14.2. The 8224 contains a crystal controlled oscillator circuit, a divide by nine counter, two (high level) drivers for $\phi_1$ and $\phi_2$ cand other logic circuits. The oscillator rcuit derives its basic operating frequency from an external series resonant, fundamental mode crystal. Two inputs are provided for the crystal connections (XTAL 1 and XTAL 2). The selection of crystal frequency depends on the speed at which 8080 is to be operated. Basically the oscillator frequency should be 9 times the frequency of the pulse $\phi_1$ (or $\phi_2$). Fig. 14.3 shows the $\phi_1$ and $\phi_2$ clock waveforms with an example to illustrate the selection of the crystal. Clock generator circuit consists of a synchronous divide by nine counter and associated gates to create the waveforms $\phi_1$ and $\phi_2$ (Fig. 14.3). The clocks $\phi_1$ and $\phi_2$ are buffered through the high level driver. The sync signal comingi

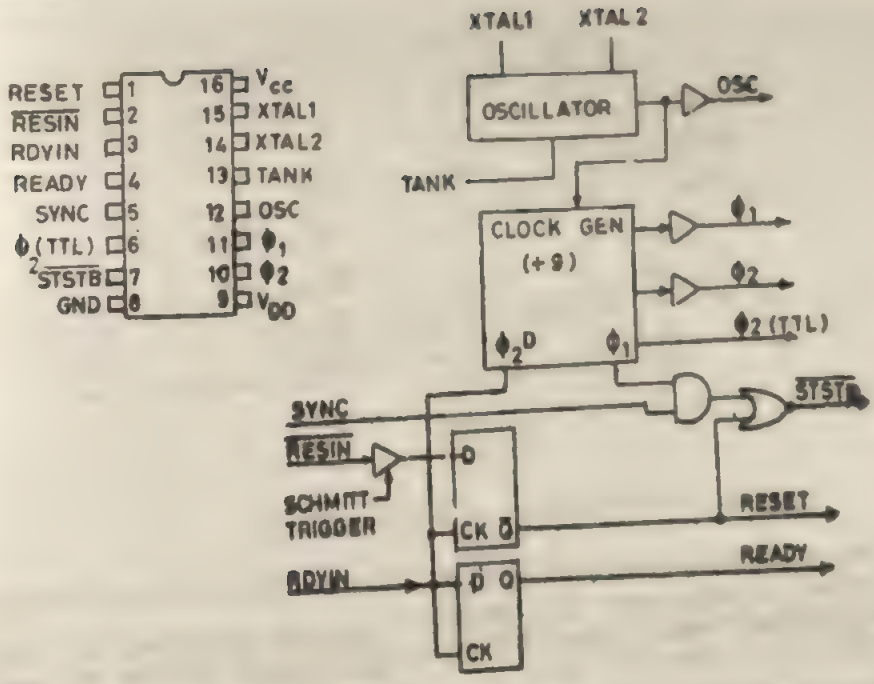



FIG. 14.2 8224 CLOCK GENERATOR

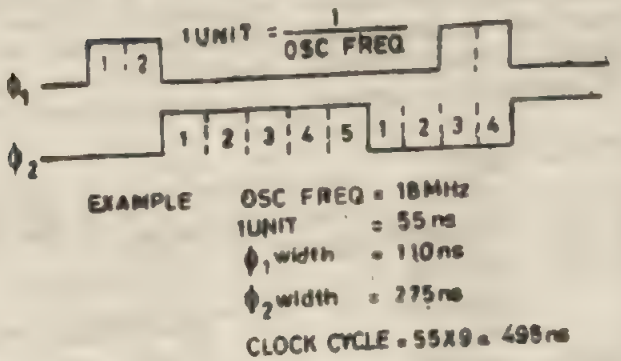



FIG 14.3 $\Phi_1$ AND $\Phi_2$ WAVEFORMS

from 8080 is gated with $\phi_1$ and is available as STSTB (status strobe) on pin 7. This signal can be used to clock the status word available on the data bus of 8080 into the external register. The asynchronous inputs RESIN (Reset In) and RDYIN (Ready in) are buffered in the flip-flops and are available as RESET and READY signals to be given to the processor. Fig. 14.4 shows the connection diagram of the 8224 with the 8080 processor chip. The circuit provided on pin 2 (RESIN) provides the automatic power ON reset, also manual reset signal can be given by the switch shown. The automatic power ON reset works as follows : When the power is put ON, the $V_{CC}$ power supply (and also other supplies) will stabilise to 5 volts, but the voltage on pin 2 reaches 5 volts much later due to RC circuit shown. When the

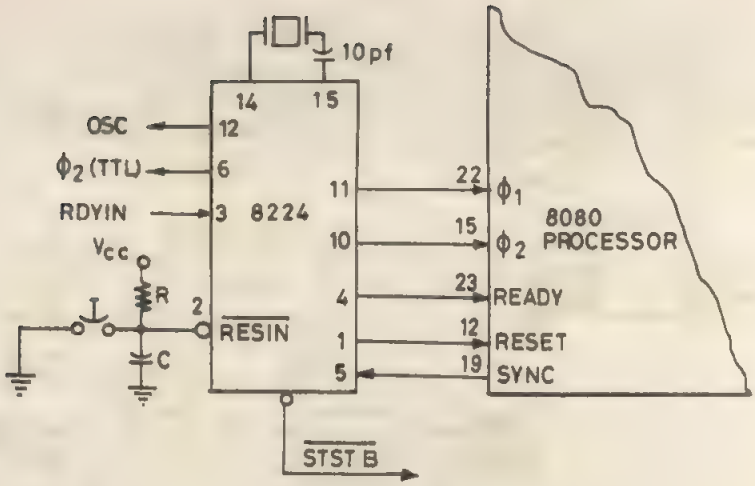



FIG. 14·4 USE OF 8224 IN 8080 BASED CPU

pin 2 charges to certain threshold value, the 8224 generates a pulse which gets latched into the rest flip-flop whose output provides the RESET signal to 8080 (Fig. 14·2).

### 8228 System Controller and Bus Driver

The 8228 is system controller and provides a buffering for the data bus. It generates all control signals required to directly interface RAM, ROM and I/O components. The block diagram of the 8228 logic is shown in Fig. 14.5.

An 8-bit bidirectional bus driver is provided to buffer the 8080 data bus. The 8080 data bus pins have the logic 1 voltage requirement of 3.3 volts (minimum) and can sink a maximum of 1.9 mA in the 0 state, thus limiting the number of chips, 8080 can drive. The 8228 data bus driver buffers the 8080 data bus and provides the drive (sink) capa-

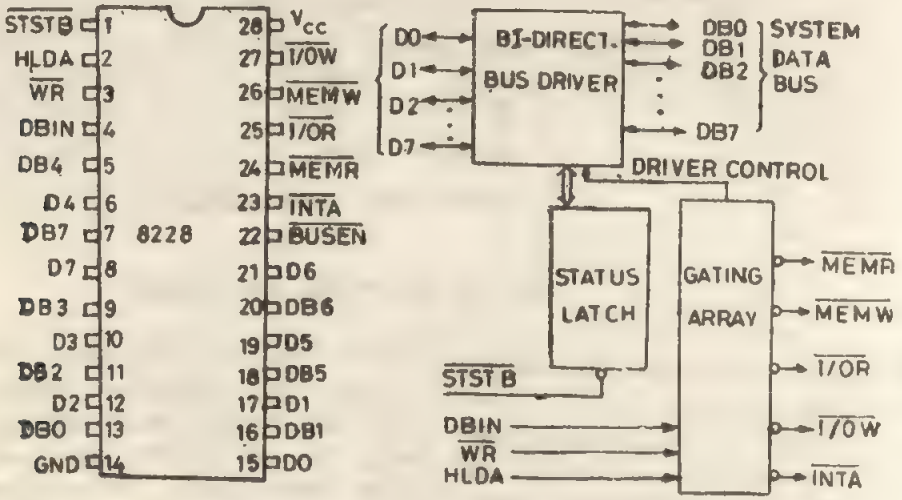



FIG 14·5 8228 SYSTEM CONTROLLER AND BUS DRIVER

| STATUS WORD | DATA BUS BIT | STATUS INFORMATION | TYPE OF M/C CYCLE: INSTRUCTION FETCH | MEMORY READ | MEMORY WRITE | STACK READ | STACK WRITE | INPUT READ | OUTPUT WRITE | INTERRUPT ACK | HALT ACK | INT ACK WHILE HALT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D0 | INTA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| D1 | $\overline{WO}$ | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| D2 | STACK | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| D3 | HLTA | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| D4 | OUT | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| D5 | M1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| D6 | IN | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| D7 | MEMR | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

FIG. 14·6 STATUS WORDS FOR VARIOUS M/C CYCLES

bility of 10 mA (typical). This allows a large number of memory or I/O components to be connected directly to the buffered data bus. The bidirectional buffered data bus (system bus) drivers are controlled by signals from the gating array.

An 8-bit register is provided on the 8228 to store the status byte which is available on the data bus. It is latched into the register by the signal $\overline{STSTB}$ given by 8224. The 8-bit register drives the gating array which decodes the status information to produce the control signals for memory and I/O devices. The gating array outputs the signals $\overline{MEMR}$

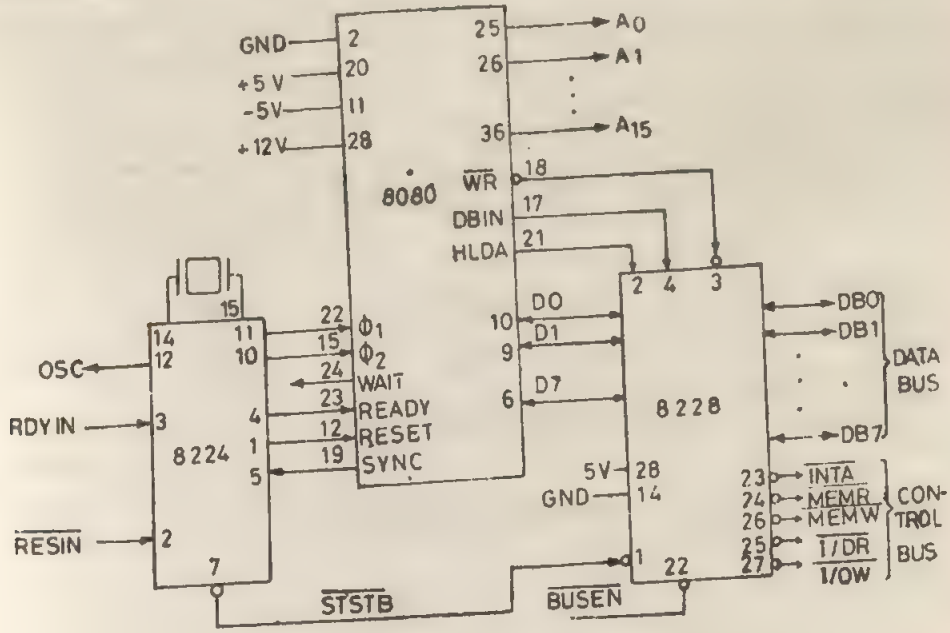



FIG. 14·7 8080 BASED CPU

(Memory Read), $\overline{\text{MEMW}}$ (Memory Write), $\overline{\text{I/OR}}$ (I/O Read), $\overline{\text{I/OW}}$ (I/O Write) and INTA (Interrupt Acknowledge) These signals form the control bus of the system. The $\overline{\text{BUSEN}}$ signal is an external signal. This signal when not active (logical 1) puts the buses (system data bus and control bus) in the high impedance state, otherwise it keeps the data and control buses in the normal working mode. The status word for various machine cycles and the control signals actually generated by the gating array is shown in Fig. 14 6, from which the logic circuits in the gating array could be easily workd out.

Complete CPU diagram using 8080, 8224 and 8228 chips is shown in Fig. 14.7. The complete CPU hardware interacting with the external devices could now be characterised by three buses, *i.e.*, address bus, system data bus aud control bus. The addition of memory and 1/O components to this hardware will make a microcomputer.

**Interfacing Memory and I/O Components to the CPU**

The memory and I/O components can be interfaced to the CPU easily through the three buses discussed. Fig. 14.8 shows the RAM and ROM interface diagram.

The 8111 is a 256×4 RAM chip with bidirectional (common I/O) data lines. The R/$\overline{\text{W}}$ line decides the read or write operation, while OD line when active disables the output drivers (in the 8111 chip), thus disconnecting the data bus from memory data lines. This allows some other device to put the data on the data bus. The 8 least significant bits of the address bus are connected to the 8 address lines of 8111 while $\overline{\text{MEMW}}$ and $\overline{\text{MEMR}}$ control signals from the control bus are connected to R/$\overline{\text{W}}$ and OD lines of 8111 respectively. Similarly 8316 ROMs can be

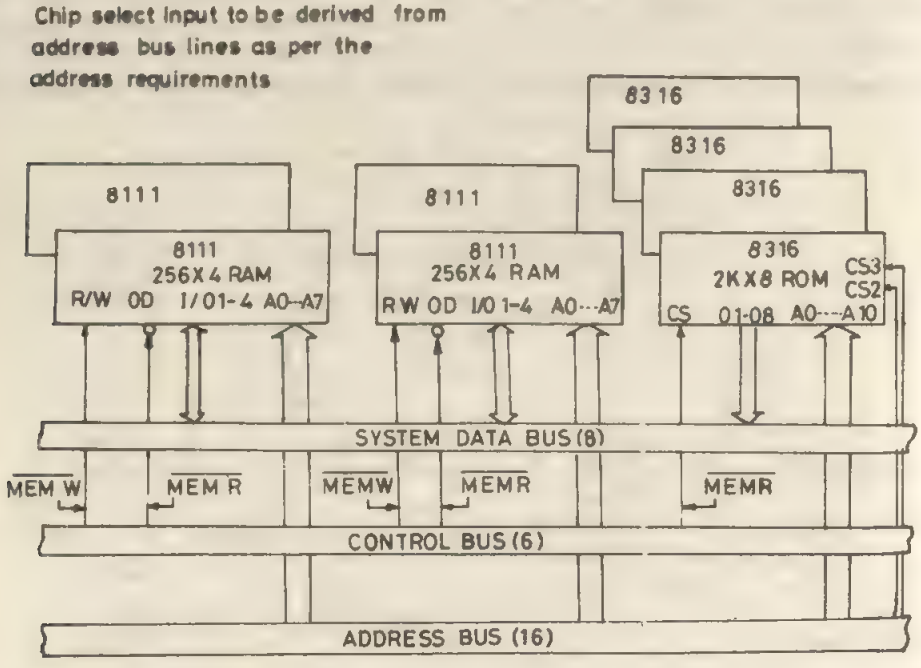



FIG. 14·8 RAM / ROM INTERFACE TO CPU

interfaced with minor variations in the control signal connections.

. An I/O device interface with the CPU has also similar connections. The buffer register of the device is to be connected to the data bus and $\overline{I/OR}$ and $\overline{I/OW}$ lines are to be used for data transfer.

## 14.3. 8085 PROCESSOR

8085 CPU chip contains more components than that on 8080 CPU chip. These additional components completely eliminate the need for 8224 and 8228 chips which are required to form a CPU based on 8080 chip. Thus a single 8085 chip, by itself is a complete CPU. 8085 is upward compatible with 8080 architecture, *i.e.*, the machine language and other facilities in 8080 are the subset of 8085 machine language and other facilities. 8085 has more interrupt sources unlike 8080 and also it has two additional instructions. Thus any program running on 8080 can be run on 8085 based microcomputer but the 8085 chip is not hardware compatible with 8080 chip.

### 14.3.2. 8085 Hardware Architecture

The block diagram of the 8085 CPU chip is shown in Fig. 14.9. We shall now discuss the functional pin description of 8085 chip.

**$A_{15}$, $A_{14}$...$A_8$ (Address Bus)**

These lines are the most significant byte of the 16-bit address bus.

**$AD_7$, $AD_6$ ... $AD_0$ (Address/Data Bus)**

These lines carry the lower order byte of the address or data byte. These are bedirectional lines During the first clock pulse of every machine cycle, these lines carry the lower significant 8 bits of the address (other 8 bits come on the $A_{15}$, $A_{14}$,... $A_8$ lines). At other time instances these lines carry the data and thus act as the bidirectional data bus.

**ALE**

The ALE (Address Latch Enable) signal occurs during the first clock of every machine cycle. It indicates that the lines $AD_7$, $AD_6$... $AD_0$ contain the address byte. The ALE signal is usually used to latch the least significant bits of the address into the external 8-bit register.

**$S_0$ and $S_1$**

These lines indicate the operation for the data bus. The codes and their meanings are as follows :

| $S_1$ | $S_0$ | |
|---|---|---|
| 0 | 0 | HALT |
| 0 | 1 | WRITE |
| 1 | 0 | READ |
| 1 | 1 | FETCH |

The $S_1$ line can be interpretted as R/$\overline{W}$ line in all bus transfers.

**$\overline{RD}$**

This line indicates that the data bus is available to read the data from memory or I/O components.

**$\overline{WR}$**

This signal indicates that the data on the data bus is to be written into the memory or an I/O port.

**READY**

If the READY line is high during a read/write cycle, it indicates to the CPU that memory or peripheral is ready to send or receive the data. If READY is low, CPU will wait for READY to go high before completing the read or write cycle.

**HOLD and HLDA**

Hold signal when true causes the $\overline{RD}$, $\overline{WR}$, $\overline{I/OM}$, ALE and all the bus lines to go into the high impedance state. HLDA acknowledges the HOLD request, *i.e.*, when HLDA is active the buses and control lines could be assumed to be in the high impedance state.

**INTR and $\overline{INTA}$**

It is used as a general purpose interrupt line. If this line is active, PC will be inhibited from being incremented and $\overline{INTA}$

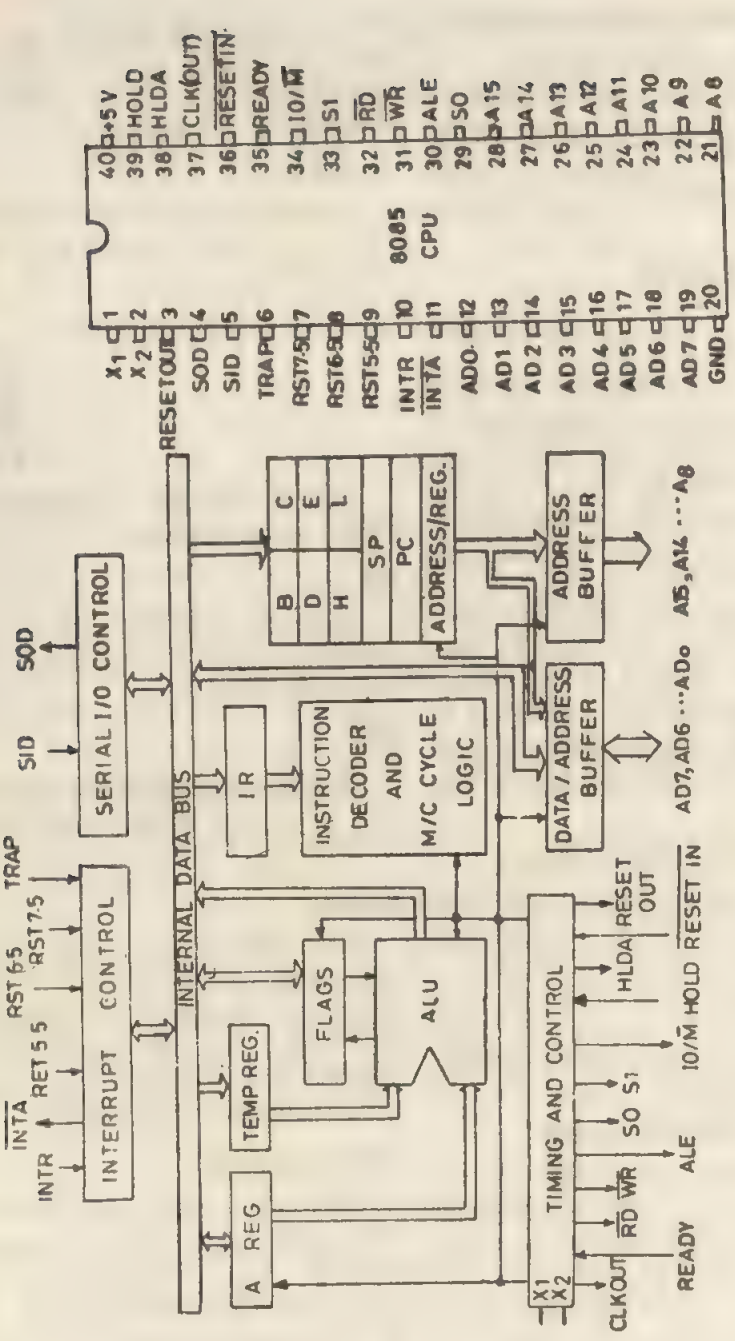



FIG. 14.9 8085 MICROPROCESSOR HARDWARE ORGANISATION

(Interrupt Acknowledge) will be issued to insert the CALL or RST instruction on to the data bus.

**RST 5.5, RST 6.5, RST 7.5 and TRAP**

RST 5.5, RST 6.5 RST 7.5 and TRAP are the four additional interrupt inputs available on 8085 processor. The priorities and the interrupt locations of these interrupts are as follows :

| Interrupt Line | | Interrupt Location |
|---|---|---|
| TRAP | highest priority | $24)_{16}$ |
| RST | 7.5 | $2C)_{16}$ |
| RST | 6.5 | $34)_{16}$ |
| RST | 5.5 | $2C)_{16}$ |
| INTR | lowest priority | Instruction to be supplied by the device. |

The interrupt due to TRAP is unmaskable, *i.e.*, it cannot be disabled, while other interrupts can be enabled or disabled.

**$\overline{\text{RESET IN}}$ and RESET OUT**

The $\overline{\text{RESET IN}}$ signal resets the 8085 CPU to the initial state. It clears the program counter, Interrupt Enable and HLDA flip-flops. None of the other registers or flip-flops are affected. The CPU remains in the HOLD condition as long as this signal is active (logical 0). The CPU starts the instruction execution as soon as this signal is made 1. The RESET OUT signal acknowledges the RESET IN, *i.e.*, it indicates that the CPU is being reset. The RESET OUT signal can be used as a reset signal for the other components of the system.

**$X_1$ and $X_2$**

A crystal or RC network can be connected on these pins for the internal clock generation. If an external clock is to be used, it should be connected to $X_1$.

**CLK**

The internal clock of the CPU is available on the line CLK. This can be used as a system clock.

**$\overline{\text{IO/M}}$**

The $\overline{\text{IO/M}}$ line indicates whether the read/write operation is with a memory or I/O device. This line goes to high impedance state during HOLD or HALT mode.

**SID and SOD**

The SID (Serial Input Data) SOD (Serial Output Data) lines provide a serial communication link between the 8085 CPU and a serial I/O device like a teleprinter or a CRT terminal. The serial data transmission is done through the bit 7 of the A register, by executing SIM or SOD instructions (to be discussed shortly).

**$V_{cc}$ and $V_{ss}$**

$V_{cc}$ and $V_{ss}$ are +5 V and ground lines. The 5 V supply is to be connected between these lines.

**MCS-85 System**

Minimum configuration of 8085 based microcomputer can be built using a small number of chips. The block diagram of such a system is shown in Fig. 14.10. Here we use 8-bit register to latch the 8 most significant 8 bits of address from Address/Data bus. The clocking of the data into the register is done by the ALE signal. This 8-bit address along with the 8 address bus lines form a 16 bit address. The 16-bit address can address the standard memory chips (A memory chip designed for 8085 based system has the above 8-bit address latch within the memory chip itself). The control signals $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{IO/M}}$ and READY give the control signals for communication with the I/O or memory devices as per the discussion given earlier for these signals.

### 14.3 3. Architecture

As mentioned earlier 8085 processor is architecturally upward compatible with the 8080 processor. In addition to the machine language instructions of 8080, 8085 has two more instructions. These are discussed in this section. For other instructions, the section on 8080 processor may be referred.

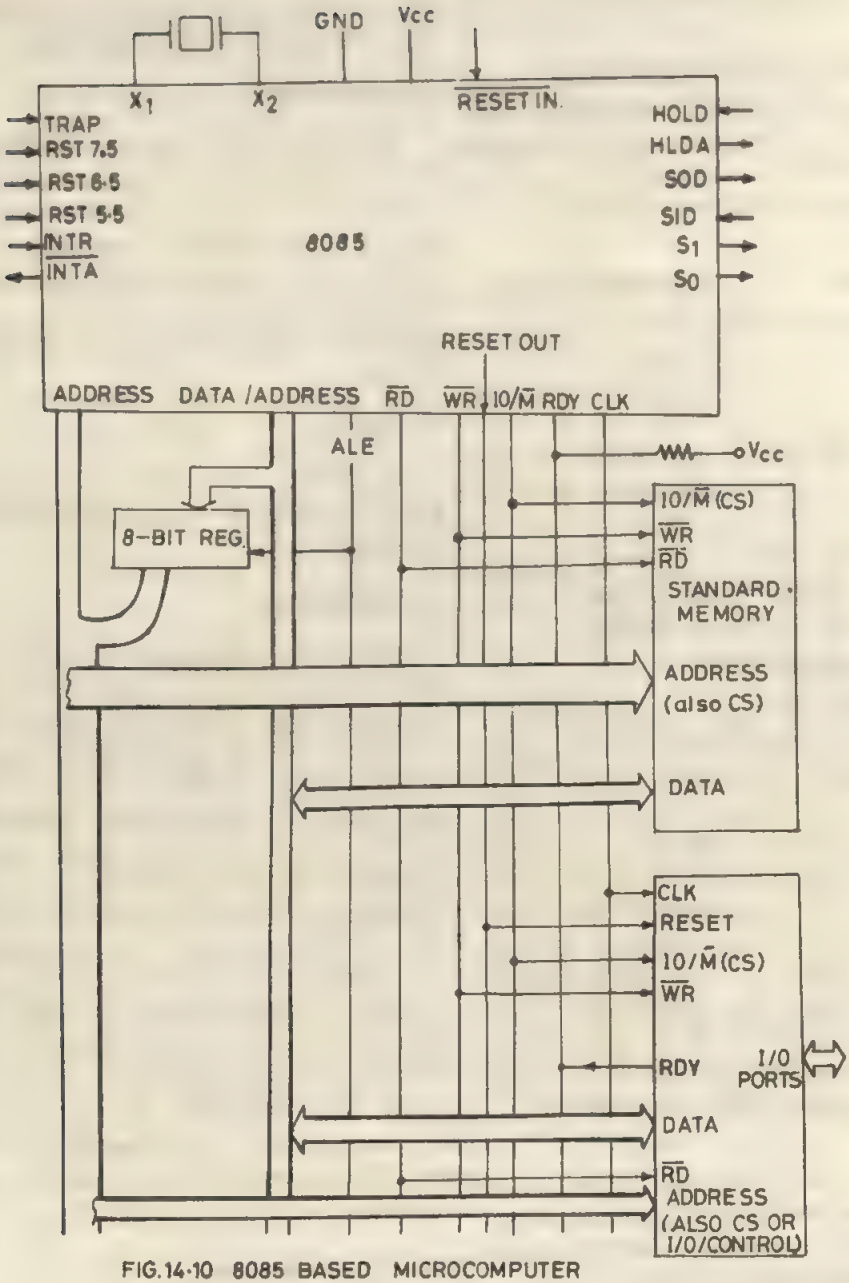



FIG. 14·10 8085 BASED MICROCOMPUTER

**RIM and SIM Instructions**

Besides the 8080 machine instructions, RIM (Read Interrupt Mask) and SIM (Set Interrupt Mask) are the two additional instructions available on 8085. These instructions interact between the accumulater and the interrupt flag, masks and SID and SOD pins of the 8085 with A register bits' correspondance as follows for SIM instruction :

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| OD | SOE | X | R 7.5 | MSE | M 7.5 | M 6.5 | M5.5 |

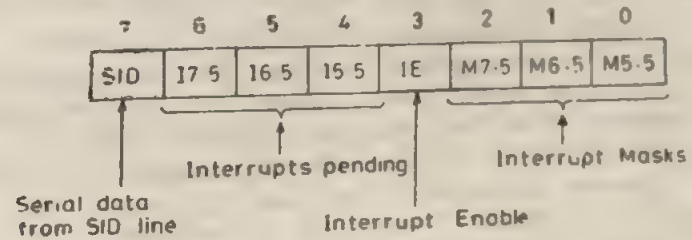
**Abbreviat s**

SOD : Serial Output Data
SOE : Enable Serial Output Data
X : Unused
R 7.5 : Reset RST 7.5 Interrupt
MSE : Mask Set Enable
M 7.5 : Mask Bit of RST 7.5 Interrupt
M 6.5 : Mask Bit of RST 6.5 Interrupt
M 6.5 : Mask Bit of RST 5.5 Interrupt.

The bits 0, 1, 2 form the mask bits for the interrupts RST 5.5, RST 6.5 and RST 7.5.

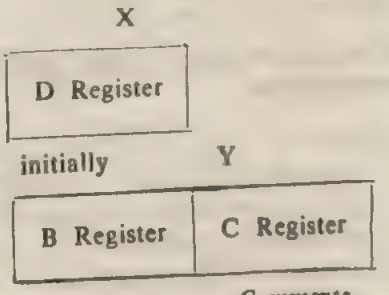
The execution of the SIM instructions causes the data from the accumulnter to be transferred to the mask register. The interrupts are disabled if MSE (bit 3) is 1 and the corresponding interrupt mask bits are made 0. If the MSE bit is 0, the interrupt mask flip-flops cannot be changed by SIM instruction. The 7th bit of accumulator appears on the SOD line if SOE (bit 6 of the A register) is also true.

The execution of RIM instruction loads the interrupt and other flag masks in the accumulator in the format shown below :




## 8080/8085 SAMPLE PROGRAM

Based on the Algorithm 7.1 given in Chapter 7, a multiplication routine is presented to multiply two 8-bit unsigned integers. The configuration of registers used in the algorithm is as follows :




| *Address* | *Code* | *Label* | *Opcode Mnemonic* | *Operands* | *Comments* |
|---|---|---|---|---|---|
| 0000 | 26 08 | MULT | MVI | H, 08 | H←8 |
| 0002 | 3A 20 00 | | LDA | X | D←X |
| 0005 | 57 | | MOV | D, A | |
| 0006 | 3A 21 00 | | LDA | Y | C←Y |
| 0009 | 4F | | MOV | C, A | |
| 000A | 06 00 | | MVI | B, 00 | B←0 |
| 000C | 79 | RPT | MOV | A, C | If LSB of Reg. C=0, then go to SHIFT |
| 000D | E6 01 | | ANI | 01 | |
| 000F | CA 15 00 | | JZ | SHIFT | |

| *Address* | *Code* | *Label* | *Opcode Mnemonic* | *Operands* | *Comments* |
|---|---|---|---|---|---|
| 0012 | 78 | | MOV | A, B | B←B+D (*i.e.*, |
| 0013 | 82 | | ADD | D | add X to partial |
| 0014 | 47 | | MOV | B, A | product) |
| 0015 | 78 | SHIFT | MOV | A, B | Shift BC |
| 0016 | 1F | | RAR | | right |
| 0017 | 47 | | MOV | B, A | by 1 |
| 0018 | 79 | | MOV | A, C | |
| 0019 | 1F | | RAR | | |
| 001A | 4F | | MOV | C, A | |
| 001B | 25 | | DCR | H | Decrement H |
| 001C | C2 0C 00 | | JNZ | RPT | |
| 001F | 76 | | HLT | | |
| 0020 | | X | DS | 1 | Declare 1 byte |
| 0021 | | Y | DS | 1 | Declare 1 byte |



The program has three parts. The first part initialises the registers. In the second part, the least significant bit of register C is checked and if it is 1, X is added to the partial product (B register). After this, the BC register pair is shifted right by 1 place. The last part of the program decrements H register by 1 and repeats the program from label RPT. After 8 steps, the program halts, leaving the product in register pair BC.

## 14.4. INTRODUCTION TO 8086 MICROPROCESOR

The 8086 is a single chip 16-bit microprocessor developed by Intel Corporation recently. It has a large number of instructions and other architectural facilities. The hardware architecture of 8086 processor allows to build a multiprocessing system incorporating a number of 8086 processors. In effect 8086 based microcomputers can be designed with the capabilities far exceeding those of conventional minicomputers. The earlier processors (8080/8085) due to their low cost were ideal for on-line process control and other industrial applications (although they could also be used for larger computational work), the 8086 provides a low cost microcomputer system with high computational capabilities (8 to 10 times that of 8085) required in a data processing or scientific computational environment.

Since 8086 has advanced architectural facilities, it is out of the scope of this book to discuss them in detail.

The 8086 addresses the main memory through the concept of segments. Each segment is 64K long. A location in a segment is addressed by 16-bit address quantity (either displacement or a contents of a 16-bit register). There can be 4 active segments at any time. They are (*i*) program segment which contains the program, (*ii*) data segment, (*iii*) stack segment and (*iv*) extra segment. There are 4 segment registers. The starting address of a segment is given by 16 times the contents of a segment register. The 8086 has also index addressing. There are number of other registers and they could be used as 8-bit or 16-bit registers. The 8086 has facilities to move data from one memory area to another, compare the data in two memory areas and many other instructions which are normally found on a large scale CPU. A powerful computer can be built using 8086 processor chips, 8089 I/O processor chips and other memory and I/O components. The details of the chips can be found from the manufacturer's reference manual and are left to readers.

## EXERCISES

1. Discuss the use of READY signal of 8080.
2. Develop a block diagram of a DMA channel to carry out DMA functions on the 8080 based microcomputer. Show clearly the connections of address, data and control buses, and use of the HOLD signal.
3. Microprocessors could be used to carry out the variety of functions. Develop the required hardware and software to implement the switching function

$$f=X+yz+wy$$

(**Hint** : develop I/O port logic to read the variables $x$, $y$, $w$ and $z$, and then write a program to evaluate the function $f$ and output $f$ on the I/O port).

What changes will be required in the design if you were to implement $f=x'+wx'y+zw$.

4. It is required to have a front panel for 8080 based microcomputer discussed. Design the front panel logic to carry out the following functions :
   (*i*) Reading of any memory location on the panel display (use LEDS for display).
   (*ii*) Writing a data into any memory location from the panel switches.
   (*iii*) Facility to execute instructions in a step by step (execute one instruction at a time) mode by pressing a switch (named instruction).
   (*iv*) Facility to execute one machine cycle of the instruction by pressing a switch (named single cycle).
5. Using 8085 based microcomputer develop a block diagram of a data acquisition system, which can read the analog data from an analog channel.
6. Repeat (6) for 8 analog channels.
7. Develop hardware and software for connecting a teleprinter to the 8080 based system (teleprinter details are given in Chapter 13).
8. Repeat (7) for 8085 based system :
   (**Hint** : use SOD and SID lines).
9. Write a program to read/print data from/to the teleprinter.
10. Design the logic circuits using conventional ICs to implement the functions of 8224 and 8228 chips.
11. A 8085 based microcomputer is to be used in a data acquisition system. One of the problems encountered is given below.

    There are three sources A, B and C from which signals are to be digitised and read in the memory.

    Signal A occurs every 0·2 to 2 ms.
    Signal B occurs every 1·5 to 5 ms.
    Signal C occurs every 4·5 to 10 ms.

    Explain how to slove the above problem with the help of block diagram using 8085 based microcomputer. Show all the I/O port connections. Draw the flow-chart for the software part of the problem.

# 15

# MICROPROCESSOR I/O INTERFACING

## 15.1. INTRODUCTION

The need for a parallel data transfer between the CPU/Memory subsystem and out side world arises in every situation where a microprocessor is used. The system designers need devices which could be easily interfaced to the system while also keeping the chip count to a minimum. Many of the I/O chips are designed to provide specific application dependant functions with a programming flexibility to cater for variations in the functional parameters. Also some devices are designed to provide very basic primitive I/O capability, CPU implementing by program the rest of the functions for which the device is to be used. This is cost effective in the low end applications. Many such chips have large number of I/O pins for usage. Since these devices are general purpose (not supporting any specific function), they could be used in variety of situations. We shall discuss the basic I/O requirement at lowest level of the system I/O, *i.e.*, how data transfer could be carried out at circuit level.

## 15.2. BASIC I/O MODES AT CIRCUIT LEVEL

A simplest I/O function car be thought of as depositing or taking out data by the CPU from I/O registers as shown in Fig. 15.1. In Fig. 15.1 (*a*), the data is simply deposited by the CPU in the output register. The register

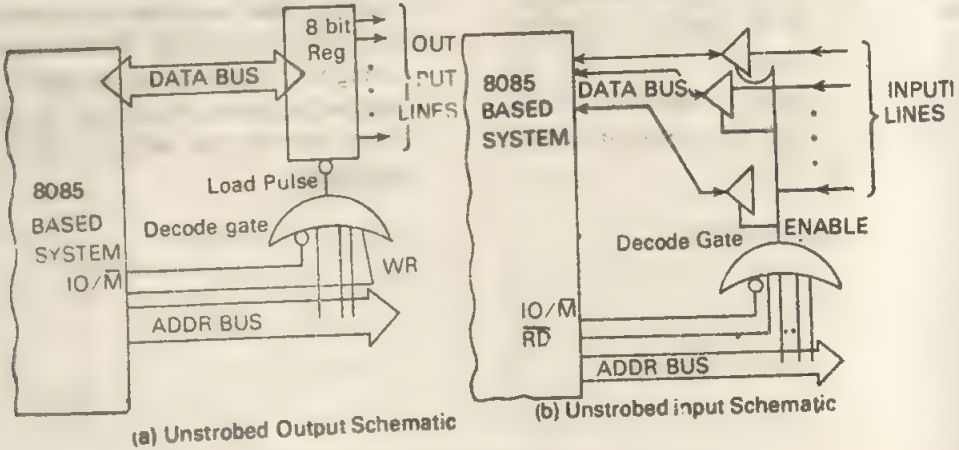



FIG. 15.1. UNSTROBED INPUT OUTPUT

output appears on the output pins. Similarly the data read by CPU is put on the input port lines. In both these cases, neither CPU (when outputing) nor peripheral device informs of data ready condition. Data lines are changed without the knowledge of other device. We shall call this mode as simple I/O (also called as unstrobed I/O). This mode is easiest to program and data transfers are done usually in synchronism with some external devices like timers etc. Timers deciding when the data transfer should be made. Examples of such I/O are programmed keyboard encode function, driving a ladder network of DAC to produce analog output, certain control settings and status reading (reading status of some switches forming jumpers for addresses of terminals (or other devices).

In contrast to the unstrobed I/O, another way of I/O which is very common is strobed I/O in which whenever data update is made, this action of updating is informed on a separate line used for the purpose. For example, when CPU puts the data in the output buffer, a flag (flip-flop) is set to indicate this condition. Let us call this flip-flop as Output Buffer Full (OBF). This signal informs the peripheral device of the incoming data. It is expected that peripheral takes away this data and informs the output port by a return signal acknowledging the data reception by it.

Upon receiving the acknowledge (ACK) signal OBF is made inactive so that the next data byte could be outputed by the CPU. Fig. 15.2 shows the logic schematic of the strobed input as well as output operations. Input operation requires a peripheral device to put the data on the port input lines, this condition is told by the signal STB which goes

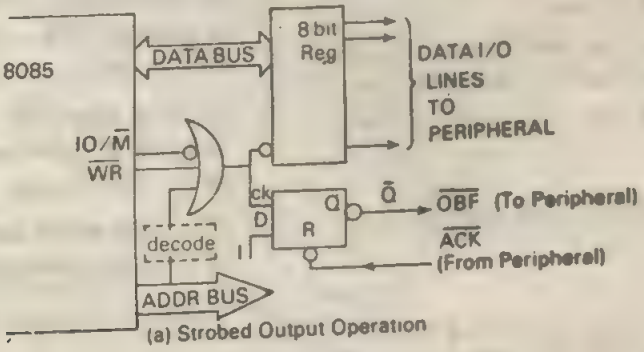



FIG. 15.2. STROBED INPUT/OUTPUT OPERATION

active whenever a new piece of data is put. The input port uses $\overline{STB}$ as a loading clock and thus latches the data into its register, also $\overline{STB}$ is used to set the flag called as IBF. This goes as a return signal to the device, which may be used for removing $\overline{STB}$. CPU will look at IBF and if true execute an IN instruction to get the data from the port into its register.

In yet another mode, I/O could be done as bidirectional bus. It may be useful in situations where a separate I/O bus is desired to be connected to a number of I/O devices to the I/O interface. Here I/O pins will have capability to handle data in both the directions as per the handshake signals originated by the peripheral. Usually the data direction and control of the bus (bus master) is in the hands of the peripheral and not the I/O interface circuit. Fig. 15.3 shows the descrete logic diagram of such an I/O mode.

When an I/O bus does **the I/O operation,** the following events occur. They shall be separately described for input and output operations.

**Output**

1. CPU Checks if output buffer is full (OBF).
2. If $\overline{OBF}$=1, (Output Buffer Empty) then CPU writes the data into the peripheral interface output buffer.
3. The $\overline{OBF}$ signal goes to the device and its activeness means a new data is there in the output port (buffer).
4. Peripheral upon $\overline{OBF}$ asserts $\overline{ACK}$ (saying it is ready to receive the data). The interface uses $\overline{ACK}$ to enable its output tri-state drivers.
5. Thus data is available on I/O bus and the peripheral may take it.
6. This ends one output I/O cycle on the bus. (Note that it is the peripheral and not the I/O interface which controls the bus).

**Input Operation**

1. The device puts its data on the I/O bus by asserting $\overline{STB}$. This enables its tri-state drivers.
2. Interface uses $\overline{STB}$ to strobe in the data into its input register. $\overline{STB}$ also sets an IBF flip-flop.
3. CPU looks at IBF and if 1 may execute an IN (Read) instruction.

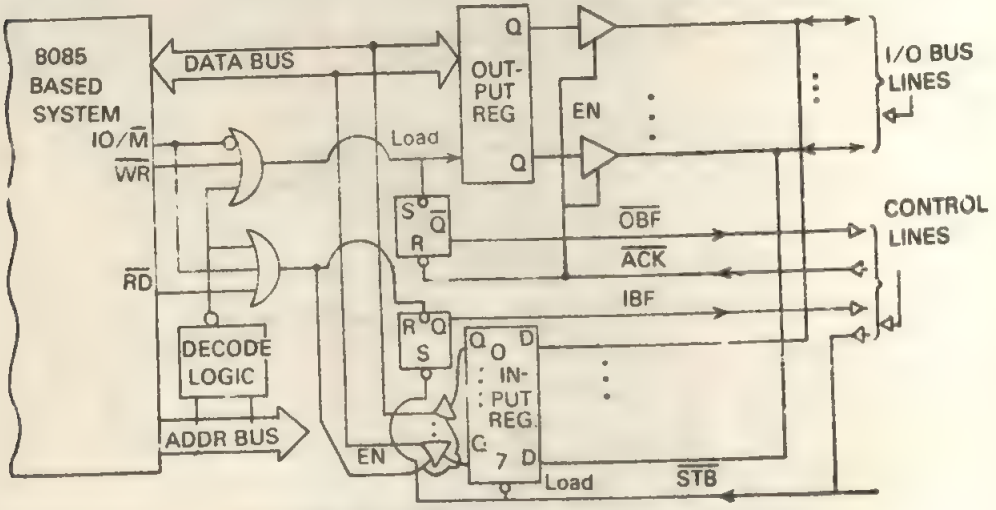



FIG. 15.3. BIDIRECTIONAL I/O BUS

4. The above action of CPU will clear IBF and transfer the data into the CPU.
5. Device may look at IBF and if 0 may initiate the next transfer.

We shall now present a general purpose I/O interface chip, 8255 called as parallel peripheral Interface (PPI).

## 15.3. THE 8255 PPI

The 8255 is a general purpose I/O interface chip having three 8-bit I/O ports thus giving I/O capability of 24 pins. Fig. 15.4 shows the basic block diagram and various functional pins. The chip has standard CPU interface lines with two address pins $A_1A_0$ to be connected to the address bus of the system. The chip thus can be accessed with 4 different port addresses to refer to the various ports and control Registers.

Ports A, B and C can operate in the basic unstrobed I/O mode and are individually programmable in mode O for input or output with additional facility that port C can be programmed with each half of it individually programmable. In strobed mode, the port acts as a control port providing control lines for ports A and B. In such a case, the chip has two

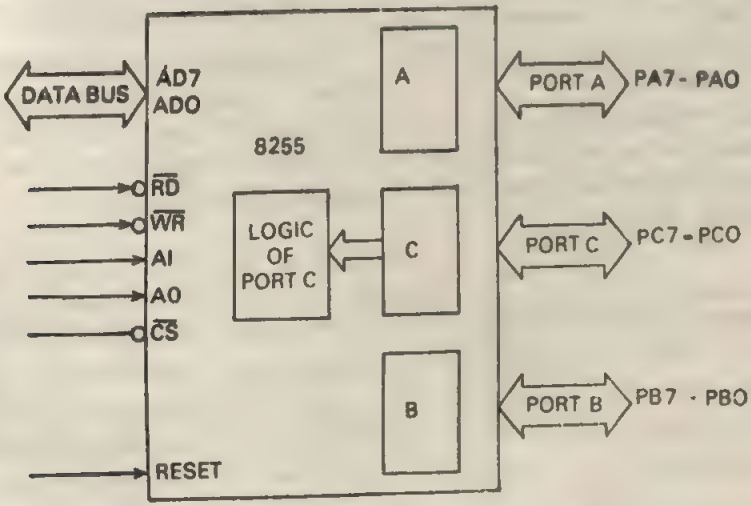



FIG. 15.4. THE 8255 PPI BLOCK DIAGRAM

TABLE 15.1. 8255 PORT ACCESSING

| $A_1$ | $A_0$ | OUTPUT OPERATION $\overline{RD}=1$, $\overline{WR}=0$, $\overline{CS}=0$ | INPUT OPERATION $\overline{RD}=0$, $\overline{WR}=1$, $\overline{CS}=0$ |
|---|---|---|---|
| 0 | 0 | DATA BUS TO PORT A | PORT A TO DATA BUS |
| 0 | 1 | DATA BUS TO PORT B | PORT B TO DATA BUS |
| 1 | 0 | DATA BUS TO PORT C | PORT C TO DATA BUS |
| 1 | 1 | DATA BUS TO CONTROL WORD REC | ILLEGAL |

groups of ports programmable in strobed mode called as group A (which includes port A as a data port and half of port C as control port for port A) and group B (here port B is a data port and half of port C acts as control). Both ports A and B has 8-bit latches (2 each), one for input and other for output. Port C has only output latch and no input latch. Bits 0 to 3 of port C (lower portion) are associated with group B, while bits 4 to 7 (upper half) are associated with group A.

To select the device for operation with the CPU, CS must be made active. The bit $A_1$ and $A_0$ of port address decides the source or destination in read or write operations from CPU. Table 15.1 summarises the data/control/status transfer operation.

In the output operation data goes to the port's output register, while in the input operation the data from input register/input lines of a port is read by the CPU.

### 15.3 1. Operational Modes of 8255

The 8255 operates in all the three modes as described earlier. All ports could be programmed in basic nonstrobed (called as mode O') input or output. Each 4 bit half of port C could be programmed individually in this mode as input or output. Fig. 15.5 shows the basic mode definition summary of 8255.

The strobed I/O mode shall be referred here as mode 1 while bidirectional I/O mode shall be referred as mode 2. Only port A can operate in mode 2. Port C bits could be set or reset individually by command word and this function shall be termed as bit set/reset.

There are two types of control words for 8255. A control word having a 1 in MSB (bit 7) sets the basic operational modes of 8255 ports, while a control word with a 'O' in MSB activates bit set/reset of port C.

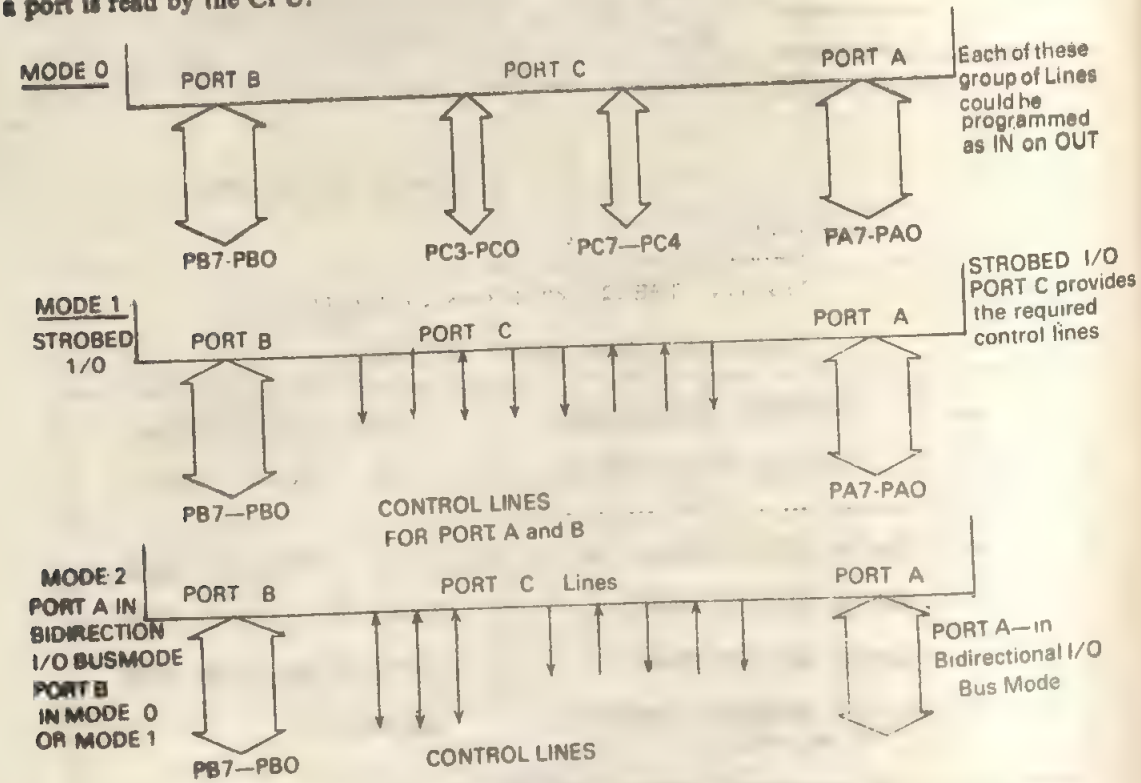
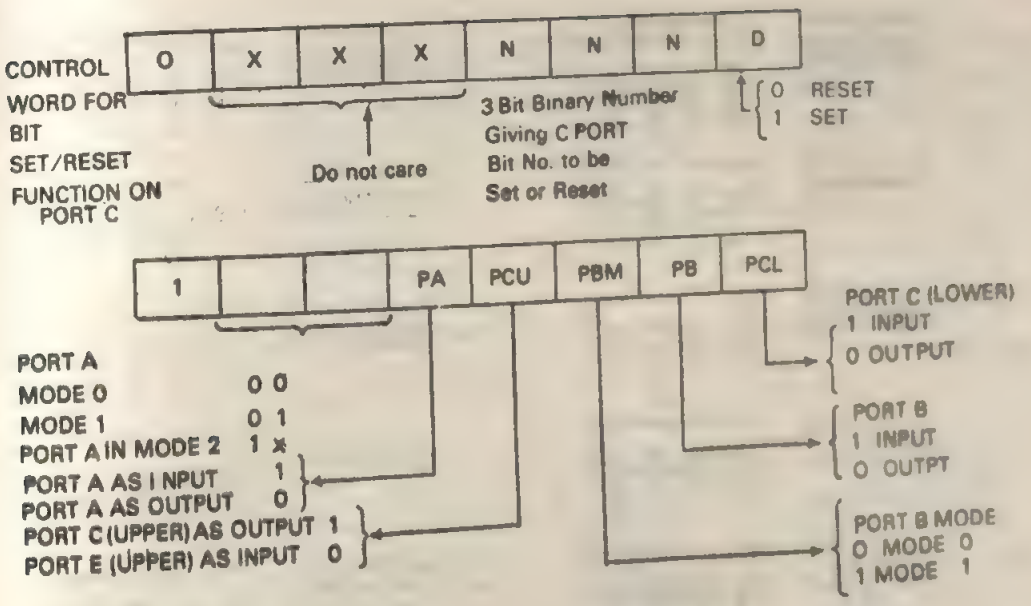



FIG. 15.5. OPERATIONAL MODES OF 8255 PPI




A control word to 8255 could be outputed if $A_1A_0$ bits of the address are both 1's. Thus if we assume that the chip is connected with $\overline{CS}$ **active for 011000XX**, then address 63 tex will access the control word register of 8255. Following program segment will do the job, of setting the mode for 8255.

```
        LDA   CW
        OUT   63H
        :
CW :    DB 1 0 1 0 1 0 1 0 B
;       CW Defines PORT A as output
        port in
;       mode 1, upper port C as input
        (always
;       in mode 0 for pins not used for con-
        trol), port B
;       in mode 0 and input mode, and
        lower port C as
;       output port.
```

**Mode 0 : Unstrobed I/O**

The 8255 has 24 I/O pins and there are 16 ways these could be programmed as shown in Fig. 15.6. The control words required are also shown.

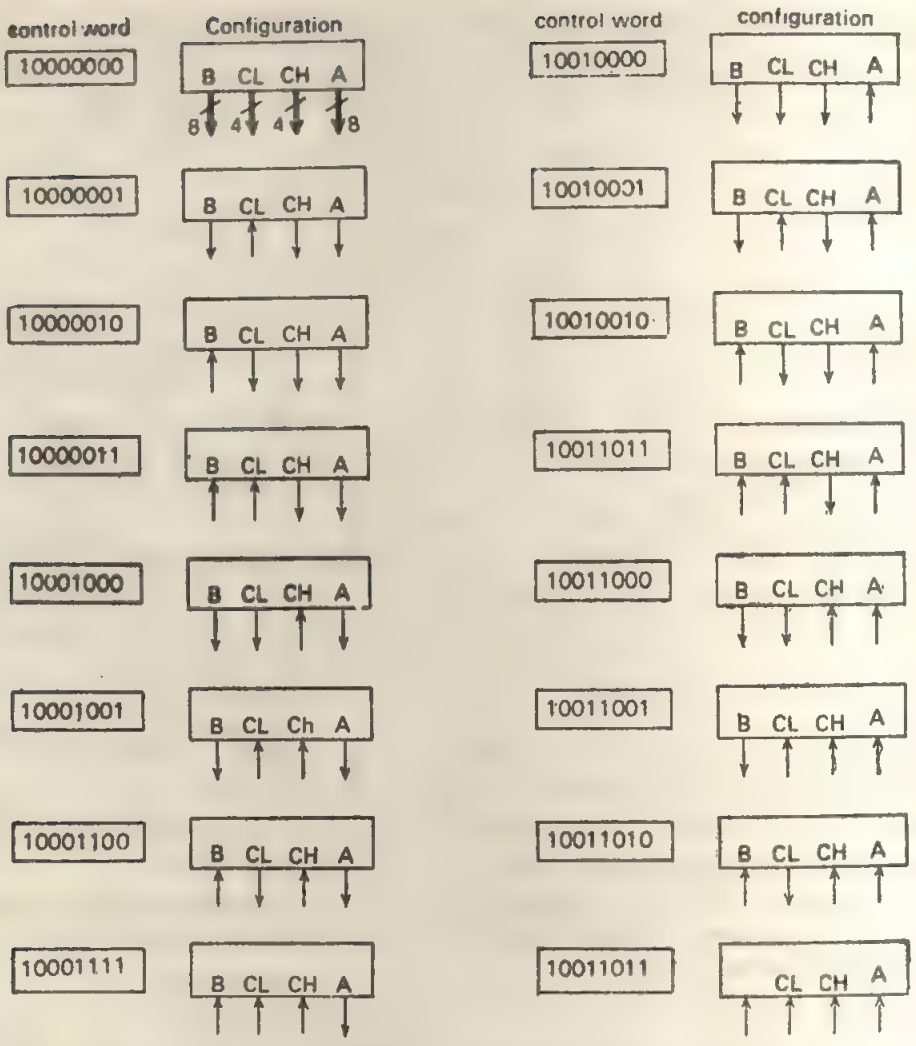



FIG. 15.6. VARIOUS MODE 0 CONFIGURATIONS OF 8255 PORTS

**Mode 1 : Strobed I/O**

Both the ports A and B could be programmed in mode 1 individually. Port C pins as designated acts as control lines and they will not operate in any mode even if programmed by control words. However, the unused port C pins could be used as per the programming done for port C. If both the ports A and B are programmed to operate in mode 1, each of them uses 3 lines each and thus only two lines of port C are unused. Fig. 15.7 shows the meaning of each pin of the 8255 port lines in various mode 1 combinations.

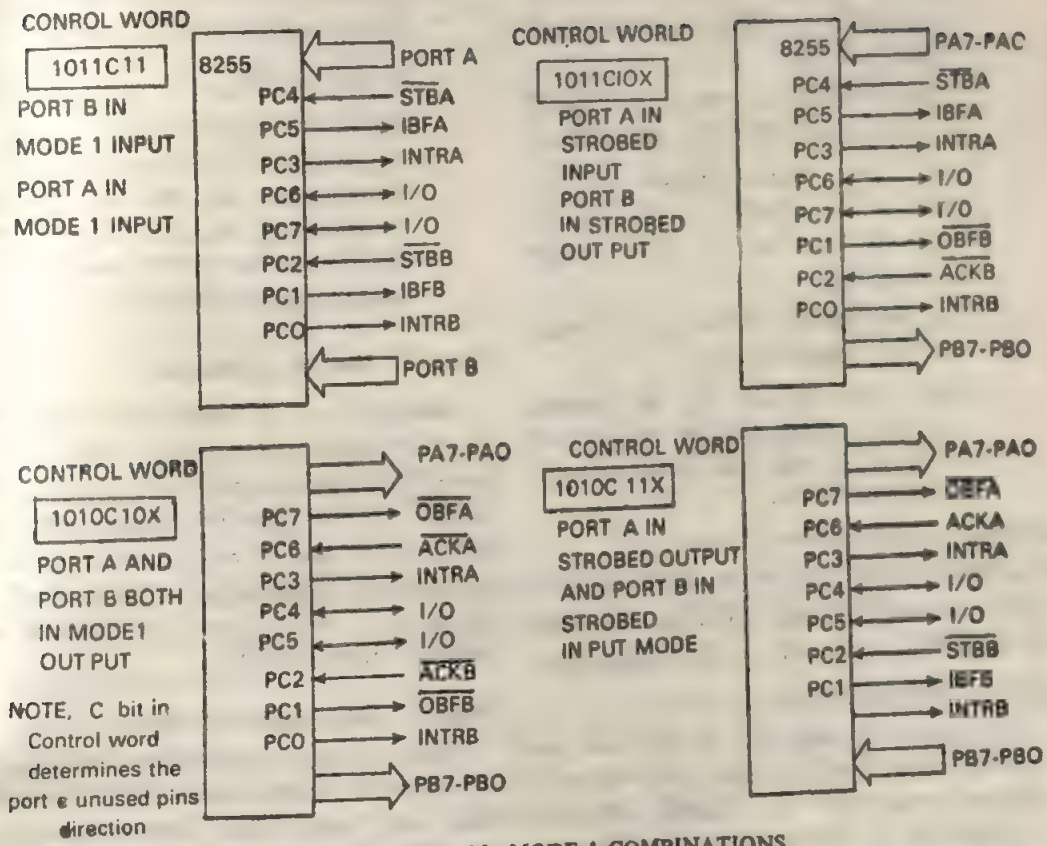



FIG. 15.7. THE 8255 MODE 1 COMBINATIONS

In mode 1, each of the ports A and B is provided with an interrupt output line. This could be used to interrupt the CPU for I/O service. In the input mode, INTR will be raised to indicate the data ready (in the input buffer) condition, while in output mode, INTR will be raised to signal the output buffer empty condition, there by requesting the CPU to output another byte.

The interrupt Enable/Disable flip-flops are also available inside 8255 for both the ports, and they are accessible as bit Set/Reset function as if they were bits of output register of port C.

FOR INPUT MODE 1
- INTE A is controlled by bit set/reset of $PC_4$
- INTE B is controlled by bit set/reset of $PC_2$

Note that a 1 is required to enable an interrupt. Executing an bit set/reset on these bits does not affect the $PC_4$ and $PC_2$ pins whatever mode port C might have been set.

FOR OUTPUT MODE 1
- INTE A controlled by bit set/reset of $PC_6$
- INTE B controlled by bit set/reset of $PC_2$

**Strobed I/O by 8255**

**Mode 1 Input by Port A**

**Actions** Pin $PC_6$, $PC_7$ are not used and they will be available as per port C (upper) programmed mode 0 I/O.

1. Data on $PA_7$, PAO is put by an external device. This is indicated by $\overline{STB}_A$ active signal put by the device.
2. 8255 uses $\overline{STB}_A$ ($PC_4$ pin) as a load pulse for its input data Register. This

causes, the strobing in of the data into PORT A. ($\overline{STB}_A$ is given on pin $PC_4$ of port C).

3. The acceptance of the data by 8255 is notified and a flip-flop called $IBF_A$ is set whose output appears on $PC_5$. State of this flip-flop is cleared to '0' by every valid read by CPU on port A in mode 1.
4. In polled mode CPU checks if IBF=1 and then reads the data.
5. $IBF_A$ may be used by the external device to transfer the next byte on port A lines, *i.e.*, next cycle must start only when IBF is low.

**Mode 1 Input by Port B**

Same as PORT A, except that port B pins carry data and $\overline{STB}_B$ and $IBF_B$ signals on pins $PC_2$ and $PC_1$ respectively are the control lines.

**Mode 1 Port A Output**

The pins $PC_4$ and $PC_5$ of port C are not required in the mode 1 protocol, they are available as normal mode 0 I/O pins. Direction is decided by the upper port C direction programmed in the control word.

**Actions**

1. In the polled mode CPU checks $\overline{OBF}_A$ for a 1 (*i.e* , output Buffer not full) by status read.
2. If $\overline{OBF}_A$=1 then CPU executes an output operation for port A.
3. The write operation on port A causes $\overline{OBF}_A$ to go low (active) indicating the output buffer full.
4. $\overline{OBF}_A$ output pin ($PC_7$) gets the value of $\overline{OBF}_A$.
5. The receiving device uses $\overline{OBF}_A$ to clock in the data into its internal register.
6. Latching of the data is indicated by the device to 8255 by $\overline{ACK}_A$ signal.
7. 8255 uses $\overline{ACK}_A$ to clear the $OBF_A$ flip-flop, thus making $\overline{OBF}_A$ to go inactive (high), to indicate the end of the current cycle, and the next cycle may be started, if required (OBF=1 means PORT A free and O means port A busy doing last operation).

**Mode 1 : Port A as Output**

Same as port A except that port B pins $PB_7$-PBO carries the data and $\overline{OBF}_B$, $\overline{ACK}_B$ and INTRB signals names and actions are done by pins $PC_7$, $PC_6$ and $PC_2$ respectively. The 8255 Ports A and B could operate in of the four possible configuration. These are shown in Fig. 15.7. All the port C pin functions may please be noted carefully.

**Mode 2 : Bidirection I/O Bus**

This mode is usually used to design an I/O system which communicates via common I/O lines. I/O system may have devices of both types input as well as output and they want to communicate through the I/O bus to 8255 and through 8255 to the CPU. Only PORT A can be configured in the bidirection I/O bus mode. Five pins of Port C are used for bus control functions. Fig. 15.8 provides with the diagram showing all the signals making port A as bus.

In the mode 2 8255 acts as slave as far as bus control is concerned. It is supposed to keep port A outputs in-tri-state unless commanded by the external device (device using the I/O BUS), All the bus activity is handled by the four handshake signals two each for one direction of the data flow. Both input and output operations on the bus shall now be presented.

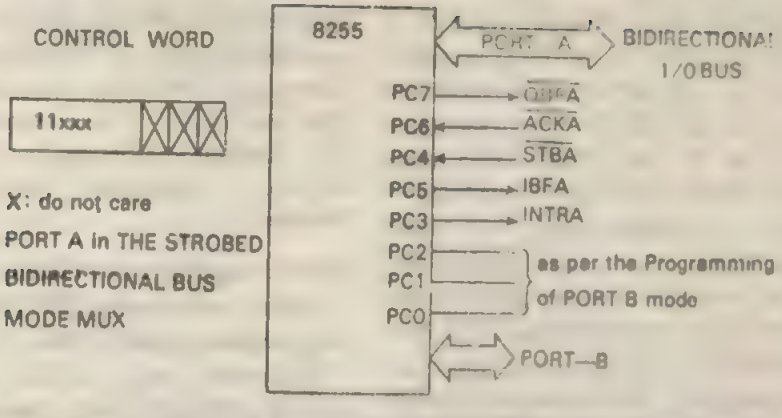



FIG. 15.8 THE 8255 PORT A IN BIDIRECTIONAL BUS MODE

**Output Operation of Port A in Mode 2**

**Actions**

1. CPU will read (in the polled mode) the flag $\overline{OBF}$ (available as a bit in the status word).
2. If $\overline{OBF}$=1 (output port finished last o/p operation) then CPU outputs the data in the port A output buffer and this sets the OBF flag making $\overline{OBF_A}$ pin active low (pin $PC_7$).
3. $\overline{OBF_A}$ signal goes to the external device to indicate that there is a new data byte in PORT A output register. Note that this data is still not on $PA_7$-PAO (these pins are still in tri-state).
4. In response to $\overline{OBF_A}$=0, device generates $\overline{ACK_A}$ active.
5. 8255 uses the $\overline{ACK_A}$ signal to enable its output tri-state drivers. This thus puts the data on the pins $PA_7$-$PA_0$.
6. Data is taken by the peripheral device and $\overline{ACK_A}$ is made inactive when $\overline{ACK_A}$ becomes inactive (changes from 0 to 1) the 8255 clears OBF flip-flop, making $\overline{OBF}$ inactive, thus ending the current output cycle.

**Input Operation of Port A in Mode 2**

Device check the condition of $IBF_A$ and if $IBF_A$ is ($PC_5$) inactive (a logical O), it initiates an input I/O cycle as follows :

**Actions**

1. Device puts its data on the lines $PA_7$-$PA_0$ by enabling its tri-state drivers by a control signal (generated by it) $\overline{STB_A}$.
2. The change in tne status of $\overline{STB_A}$ is recognised by the 8255 and it will latch the data on the $PA_7$-$PA_0$ lines into its input buffer.
3. The 8255 also sets the internal $IBF_A$ flip-flop in response to $\overline{STB_A}$.
4. This informs the peripheral device putting the data on the bus about the data reception by 8255.
5. Peripheral removes its $\overline{STB_A}$ in response to $IBF_A$ doing active (high).
6. CPU in the polled mode will read $IBF_A$ flag (by status read) and if 1 assumes the valid new data in the port A input register.
7. This read by CPU will clear IBF flag thus ending one cycle.

**Interrupt in Mode 2**

To operate the 8255 Mode 2 of port A, a common interrupt signal is provided to signal requirement of the CPU service. The pin $PC_3$ acts as $INTR_A$ will go active when IBF is high (data in the input buffer) for input data direction on the bidirectional I/O bus, or when $\overline{OBF} = 1$ (out buffer empty hence needing the new data for transmission) in the output data direction. To keep the control over INTR two separate interrupt enable (IE) flip-flops are provided for each direction and interrupt can be generated only when the relevant IE flag is set. The flags can be set/reset by bit set/reset function on the port C. The port C (only for this function) bit allocation is as follows :

$PC_6$ : $IE_1$ (Enable output direction interrupt).

$PC_4$ : $IE_2$ (Enable input direction interrupt).

$IE_1$ is associated with $PC_6$ while $IE_2$ with $PC_4$. It should be noted that these flags do not affect in any manner the operation of these pins of port C. Only $IE_1$, $IE_2$ are addressed as if they were $PC_6$, $PC_4$ respectively for bit set/reset function.

**15.3.2. Status Information in 8255 Operation**

Port C read, when other ports are programmed in mode 1 or 2 gives the status information. In mode 0, port C will transfer data to (from) CPU from (to) the peripheral connected the port C. The status word obtained when port A/B are programmed in mode other than mode 0 is as follows. There are three possible interpretations depending upon the mode of A/B ports. The bit meaning is as follows :

**15.3.3. Applications of 8255**

In low end systems, 8255 could be used for programmed keyboard display functions. Fig. 15.9 shows the usage of 8255 all operating in mode 0. The chip select signal is derived from address combination 011000XX thus giving 60H, 61H, 62H and 63H as port addressed. The 8255 having large number of I/O pins can serve as a powerful single chip solution to interfacing problems in applications where support chips are either not available or 8255 provides addition pins for extra requirements. In such applications, CPU will be executing the actual function for which the 8255 is used, 8255 providing only the necessary I/O capability.

**Key Board/Display Interfacing Using 8255**

8255 can drive a display/keyboard as shown in Fig. 15·9. Here keyboard is not standard ASCII keyboard but small application dependant.

In Fig. 15.9, the port A is configured as input port getting the 8 Return lines of the

STATUS WORDS OF 8255

| | D7 | D6 | D5 | D4 | D3 | D2 | D1 | DO |
|---|---|---|---|---|---|---|---|---|
| MODE1 INPUT | I/O | I/O | IBFA | IEA | INTRA | IEB | IBFB | INTRB |
| MODE1 OUTPUT | $\overline{OBFA}$ | IEA | I/O | I/O | INTR | IEB | $\overline{OBFB}$ | INTRB |
| MODE2 | $\overline{OBFA}$ | IE1 | IBFA | I E 2 | INTRA | X | X | X |

AS Defined for the Port B Mode

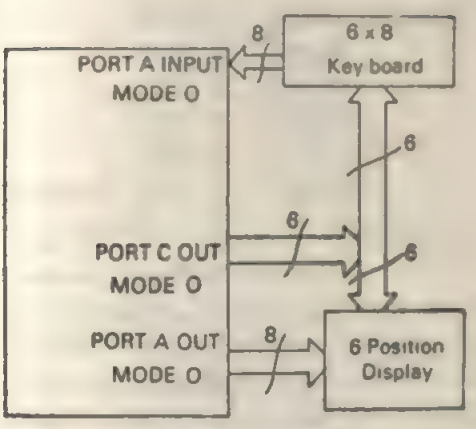



FIG. 15.9. THE 8255 DRIVING KEYBOARD AND DISPLAY

keyboard, while port B is driving the 8-bit (segment) display. Six lines from port C are used to carry out scan function. A Roting 'O' should be provided on these lines. Remaining tw › lines of the port C are one bit output lines and through the CPU program (along with a periodic interrupt), these could carry the serial data if desired or could do some control function.

RESET Function for the circuit of Fig. 15.9 is served by the Routine RESET given below :

```
RESET   LDA    CW
        OUT    63H
        RET
```

```
CW      DB     1000    1000B    set all port in
                                mode O as
                                desired
```

**Use of 8255 in Mode 1**

We shall use 8255 in a data acquisition system where there are 128 analog channels from which the data is to be acquired. We assume that channels are carrying slowly varying analog signals like temperatures, pressures and other physical quantities. Thus the system is designed with a single ADC. The hardware set up is shown in Fig. 15.10.

In the present set up, port B is used in the strobed input mode (mode 1). ADC gets a sample/hold signal from 8255 pin $PC_7$ and if actived, starts the conversion. The ADC when ends the present conversion, informs 8255 by the active low end of conversion signal. This is connected to $PC_0$ ($\overline{STB}$) which acts as strobe pulse for port B in mode 1.

Assuming usual port addresses (60H, 61H, 62H, 63H) we present a program to set up the 8255 for the desired configuration and also a program to perform scanning function to acquire data from the channels. Data acquiring is done at periodic intervals by executing an interrupt routine invoked by the timer interrupt. It is assumed that scan time necessary to

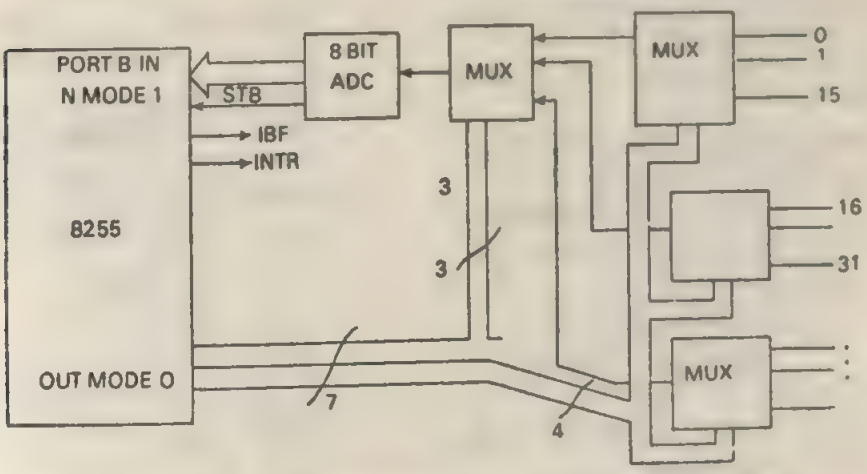



FIG. 15 10. 128 ANALOG CHANNEL DATA ACQUISITION SYSTEM.

monitor all the channels is very small compared to the period of timer.

```
; RESET PROGRAM CALLED WHEN
  SYSTEM RESET
; IS DONE
RESET     : LDA MODE CW
            OUT 63H
            RET
MODE CW   : DB  10000110 B, port
            definitions.
```

The functions of interrupt service program is to read data from every channel and store them in the locations allocated. We shall have a 128 locations of RAM allocated for the purpose.

Steps of the service routine are as follows :

1. Output analog channel address on the port A.
2. Make sample signal 1 (active).
3. Wait for ADC conversion end signal and transfer of ADC data to port B Reg.
4. Read in the port B.
5. Store the result in the location allocated for the channel.

```
; FOLLOWING PROGRAM IS CALLED
  BY TIMER INTERRUPT
; AND IT SCANS ALL THE CHAN-
  NELS
; E REGISTER IS USED FOR CHAN-
  NEL NUMBER (INDEX)
; ONLY 7 BITS OF E ARE RELEVANT
; PORT ADDRESSES ARE 60H, 61H,
  62H AND 63H FOR
; A, B, C AND CONTROL WORD
; THIS IS BECAUSE 011000XX IS
  U ED FOR CHIP SELECT
; MEMORY USED TO STORE THE
  RESULT IS CHNMEM.
TRINT : PUSH  PSW
        PUSH  B
        PUSH  D
        PUSH  H
        MVI   D, 00
        XRA   A       ch. no=0  (in
                      E Reg)
        MOV   E, A    sample enable
                      bit and
                      ch. no=0
L1 :    OUT   60H     ;
        NOP           introduced to
                      allow
        NOP           ; Mux to settle
        MVI   80H     start the
        XRA   E       conversion,
                      (keep addr. of
        OUT   60H     Mux changed
WAIT :  IN    62      ; status Read
        ANI   02      ; check D1 of
                      status word
        JZ    WAIT    ; IBF≠1
                      hence loop
; DATA IS READY AT THIS POINT AND
  MAY BE
; READ
        MOV   A, E
        OUT   60H     ; Remove
                      sample/hold
        IN    61H     } Read the
        LXI   H, CHNMEM  } data from
        DAD   D       } port B
        MOV   M, A    } and put in
                      } Memory
        INX   D
        MOV   A, E
        ANI   80 H
        JNZ   L3      ; all channels
                      read ?
        MOV   A, E
        JMP   L1
```

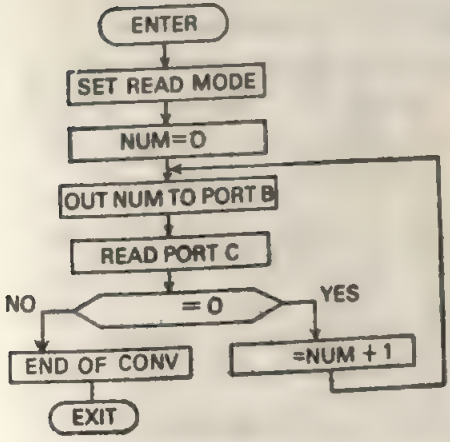
```
L2 :    POP     H
        POP     D
        POP     B
        POP     PSW
        RET
CHNMEM  DS      128
```

**Use of 8255 in Analog Output and Programmed ADC Function**

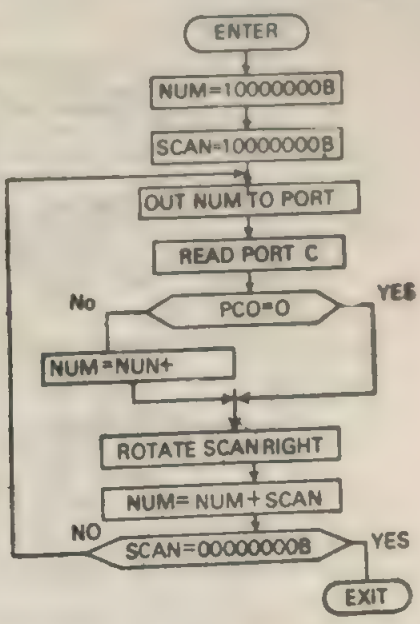
A system with a DAC output could also be converted into an analog data acquisition, by implementing ADC function in software. The hardware support needed is minimal a DAC, driven by an unstrobed output port, and a one inport port line to read in the comparator output. Comparator compares the digital number outputed (corresponding analog voltage is available as DAC output) with the analog input. An ordered search of a number which will correspond with the given input voltage is to be tried by the microprocessor. Two algorithms are presented in Fig. 15.11 (a) and (*b*) while the necessary hardware is shown in Fig. 15.12.

```
; SEQUENTIAL SCAN ADC TRIES
  NUMBERS FROM O ONWARDS
LADC :  LDA     CW
        OUT     63H
        XRA     A
L1 :    STA     NUM
        OUT     61H
        IN      62H
        ANI     01
        RNZ
        LDA     NUM
        INR     A
        JMP     L1
CW      DB      10000001B
```

Binary search algorithm, tries numbers in a better manner The first number tried is in the middle of the range, *i.e.*, Most significant bit 1 and all other bits O's. This number is outputed to DAC The comparator will produce the output 0 if Vin > VDAC, meaning that the number tried is smaller or equal to the desired number. In this case, this number should be modified to reflect the number in the middle of




(a) Sequential Search ADC




(b) Binary Search

FIG. 15.11. ADC ALGORITHMS

he half the range *i.e.*, 110 ... 0 will be next number tried. If comparator gives out a 1, it means the number tried is larger and the next number should be in the other half of the range. *i.e*, 010 ... 0 will be tried. The process is repeated till all bits are tried. Program is presented now.

```
; PROGRAM FOR SUCCESSIVE
  APPROXIMATION ADC
; B REG HOLDS THE NUMBER,
  WHILE C HOLDS SCAN
BA DC : LDA CW
        OUT 63 H
        MVI B, 10000000B
        MOV C, B
        MOV A, B
L2    : OUT 61H
        IN  62H  ; try a number
        ANI 01
        JZ  L1   Vin VDAC, go to L1
        MOV A, B
        XRA C
        MOV B, A
L1    : MOV A, C
        RAR      ; Rotate Scan
        MOV C, A
        RZ       ; exit all bits tried
        XRA B    ; append a 1 in the next
        MOV B, A ; bit
        JMP L2   go to L2 for repeating
CW    : DB  10000001B
        END
```

Fig. 15.13. shows the descrete logic implementation of both the algorithms.

## 15.4. THE 8155/8156 : RAM AND I/O

### 15.41. Introduction

The 8155/8156 is a general purpose I/O component with 256 bytes of RAM. The RAM combined with I/O ports provide a single chip solution to the microprocessor based low end applications needing I/O capabilities. A 3 chip set 085, 8155 and 8355 provides CPU, 2K ROM for program/constant storage and 256 bytes RAM for working storage for stack and other data with 22 pins of I/O capability.

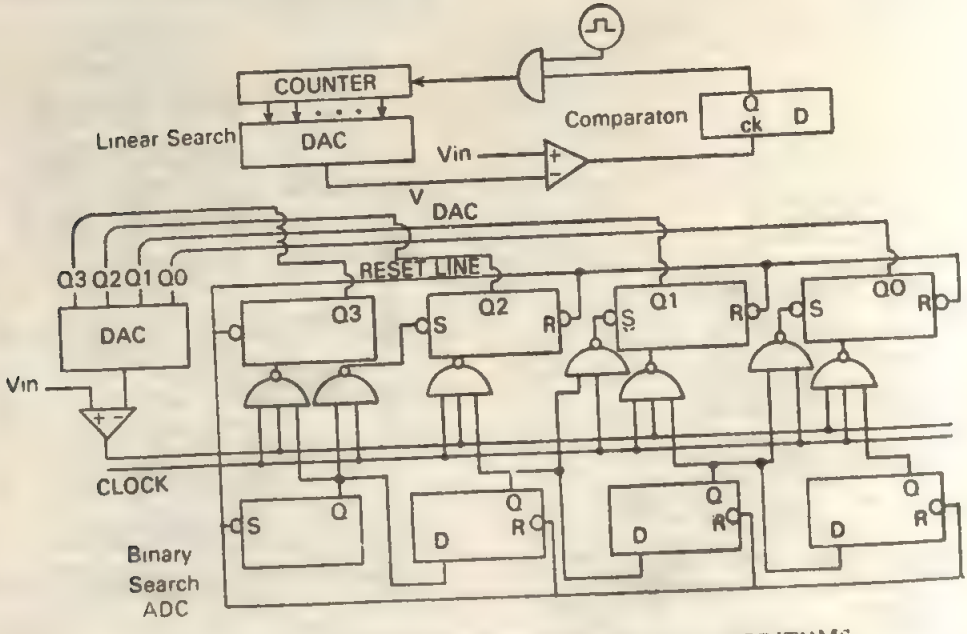



FIG. 15.12. ADC CIRCUITS IMPLEMENTING THE ALGORITHMS OF FIG. 15.12.

Besides a timer provided in 8155 could be used to serve periodic events by interrupt servicing. The present section is devoted to the study of 8155. The 8156 is identical to 8155 in functions and pin layouts except that its chip enable is active high. This allows 8155 and 8156 to be used in pair, whose chip selects are driven by a common address line without any external inverter which otherwise would have been required if both chips used were 8155s. The 8155 has a CPU interface completely compatible with the 8085 CPU, all lines to be connected pin to pin. Fig. 15.13 shows the block diagram of the chip with function pin names.

### 15.4.2. CPU Interface

The chip provides an internal 11-bit latch in which AD7-AD0, IO/$\overline{M}$ and $\overline{CE}$ lines are latched by the ALE signal. The outputs of these latches provides Address lines $A_7$-$A_{15}$ and $\overline{CE}$ and IO/$\overline{M}$ signals for the working of the memory and I/O sections. The access to I/O section thus can be done only when IO/$\overline{M}$ line is 1. Memory section is accessible when IO/$\overline{M}$ line as 0. These accesses could occur only when, the $\overline{CE}$ (internal) line is active.

The 8-bit address available internally (in the 8-bit internal latch) provides the address of the memory location in RAM if IO/$\overline{M}$ signal of

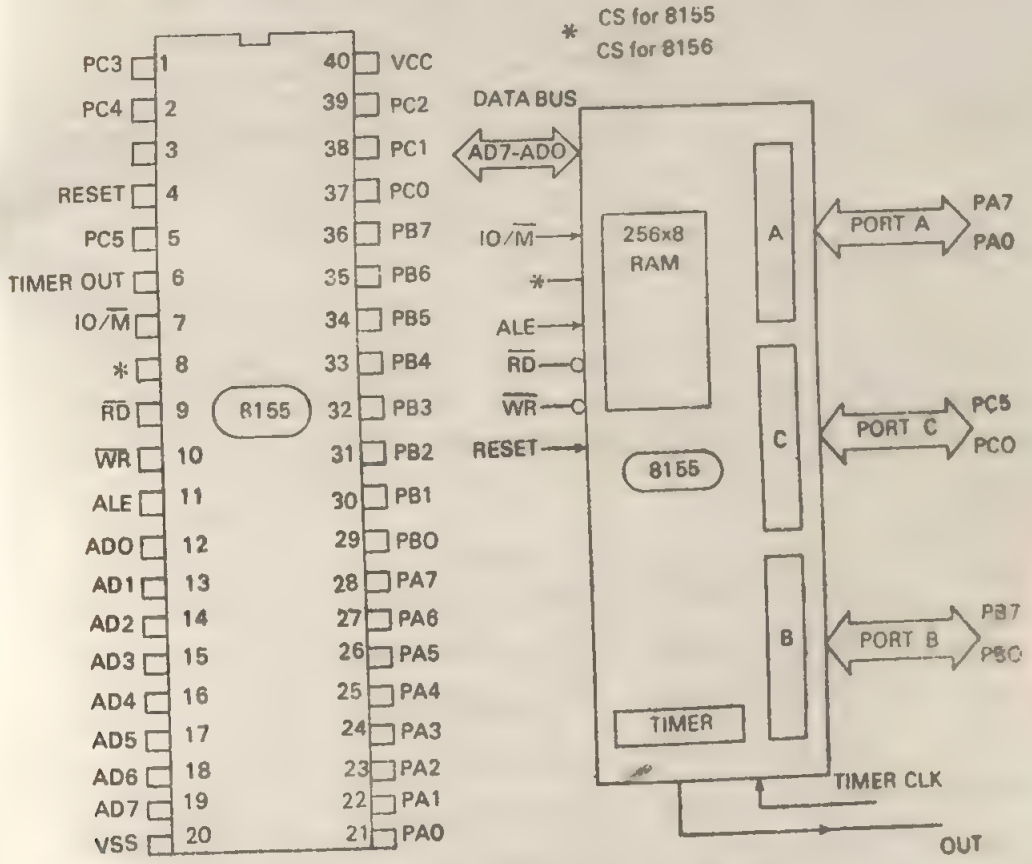



FIG. 15.13. THE 8155/8156 BLOCK DIAGRAM.

this chip is connected to IO/$\overline{M}$ signal of 8085 and shall provide separate I/O addressing *i.e.*, only IN or OUT instruction will be able to access the I/O ports. In the memory mapped I/0, one of the address line (usually $A_{15}$) is to be connected to the IO/$\overline{M}$ pin of 8155. This will give access to the chip's I/O section for every instruction which refers to the memory locations having '1' in bit $A_{15}$ and other bits generating an active chip enable signal and address for the I/0 or memory.

### 15.4.3. I/0 Section : Ports

The I/0 section of 8155 has 3 ports, PORT A, PORT B and PORT C. All these ports have programmable direction control. The PORTS A and B could be operated in the unstrobed or strobed mode with port C lines serving as handshake control signals. In the unstrobed mode all the three ports can operate independantly in the *input* or *output* mode as programmed individually. There is also a times in 8155 which has 16 bits of data transfer requirement from the CPU. Thus CPU has to address PORT A, PORT B, PORT C, TIMER HOB and TIMER LOB, and CONTROL/STATUS. This requires 3 bit address for addressing these. Following table gives the addressing summary. Note that the mentioned entity is accessible only when the chip select is active and IO/$\overline{M}$=1.

The read production ($\overline{RD}$=0, $\overline{WR}$=1 and IO/$\overline{M}$=1) reads the data from a specified port/register, while write operation puts the data in to the designated port/register. The command register gets a data byte during write operation which has the following meaning.

The ports can operate in four alternatives ALT1 to ALT4. In the alternative **ALT1, port C** operates in unstrobed **INPUT** mode Also **Ports A** and **B** operate in **unstrobed mode.** Their direction is specified by PA and.

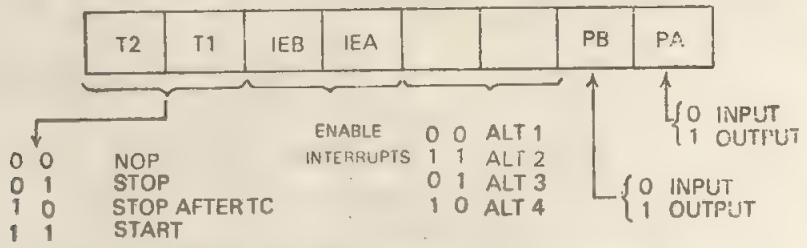



**Table II. Port C Pin Assignments**

| PORT C | ALT1 ALT2 | ALT3 | ALT4 ALT4 |
|---|---|---|---|
| PC0 | INPUT OUTPUT | INTRA (INTERRUPT) | INTRA INTRA |
| PC1 | INPUT OUTPUT | $\overline{BFA}$ (BUFFER FULL) | $\overline{BFA}$ |
| PC2 | INPUT OUTPUT | STB (STROBE FOR A) | STB |
| PC3 | INPUT OUTPUT | OUTPUT | INTRB |
| PC4 | INPUT OUTPUT | OUTPUT | $\overline{BFBB}$ |
| PC5 | INPUT OUTPUT | OUTPUT | STBB |

PB bits in the command word. The ALT2 is same as ALT1, except that the port C operates as unstrobed output port. In the ALT3, port A acts in the strobed IN or OUT as decided by PA bit in the command word. The port B operates in the unstrobed mode. The 3 lines $PC_0 - PC_2$ of the port C are used as control lines and the remaining three lines of the port C operate in the output mode. In ALT3, writing a byte into port C will not affect the pins $PC_0 - PC_2$ since they are used as control lines and thus these pins will not take part as port C lines. **In the ALT 4,** both the ports A and B operate in strobed mode and all the port C lines are used for control. In this mode any access to port C may not give any meaningful result and thus may not be tried. Particularly you should not interpret port C read data as status. Table II shows the summary of the four alternatives in which the operational details are summarised for the four alternatives as shown earlier in the Table of PORT C Pin assignment. The ports are addressed as shown below for data transfer.

Note that a Read operation on a port configured as an output port will give the port data to the CPU. Also it is to be noted that output latch is cleared when a port enters the input mode and the output latch cannot be loaded by write operation if the port is configured in the input mode.

The 8155/8156 ports could be used in a various applications like those for which the 8255 is used except that the 8255 could be used in a bidirectional bus mode with its A port.

**15.4.4. I/O Section : Timer**

The 8155/8156 timer section provides a 14 bit counter that counts TIMER IN pulses, and can provide modes like single square wave, continuous square wave, single pulse or continuous pulses at TIMER OUT pin for each terminal count (TC) as shown in Fig. 15.14. The counter timer cycle (period) is divided into two parts, first part (TIMER OUT high) and second part (TIMER OUT LOW) for each specified count length The timer data (16-bit) consists of 2 bit timer mode specification and 14 bit count length. If the count length is odd, the first half period is longer than the other half. Fig. 15.14 shows the timer out waveforms for a count of 9.

The timer in 8155/8156 is designed to count one count cycle of length CNTL by counting twice the CNTL/2 in a down counter. Fig. 15.15 shows the schematic of the logic. The programmed CNTL is halved, the LSB going to flip-flop QA and remaining 13 bits to the down counter. The QA flip-flop con-

| A2 | A1 | A0 | OPERATION OF SELECTION |
|---|---|---|---|
| 0 | 0 | 0 | COMMAND OR STATUS |
| 0 | 0 | 1 | PORT A |
| 0 | 1 | 0 | PORT B |
| 0 | 1 | 1 | PORT C |
| 1 | 0 | 0 | LOWER ORDER BYTE OF TIMER |
| 1 | 0 | 1 | Higher Order Byte of TIMER |

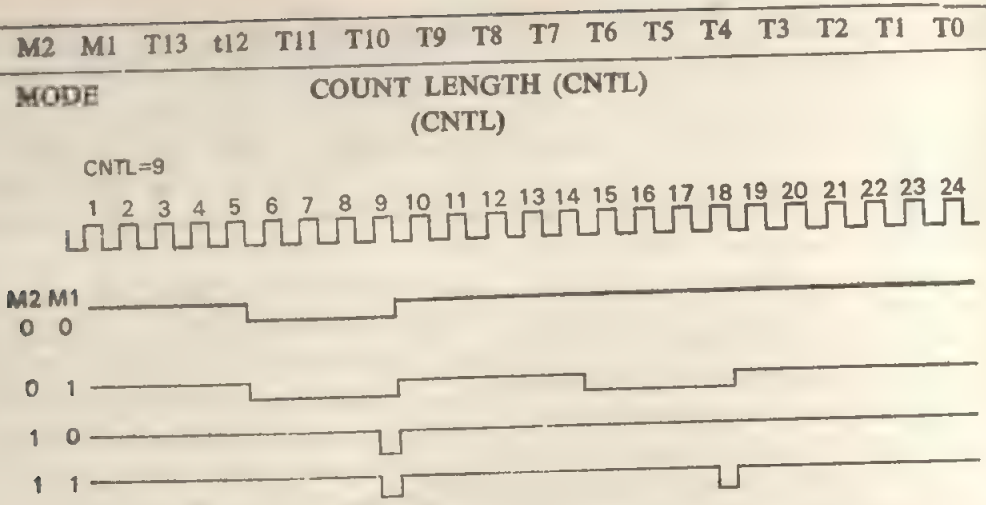



FIG. 15.14. TIMER MODES AND OUTPUT WAVEFORMS.

tents is the remainder in dividing CNTL by 2. Thus in odd CNTL the first half cycle is longer than second half cycle. The counter has a flip-flop $Q_F$ which is set to 1 normally.

When the counter generates internal TC (ITC), $Q_F$ is toggled. The first toggling of $Q_F$ indicates the end of first half count period, and the counter is again loaded with CNTL/2 ($Q_A$ is kept at 0). Now again the counter counts and will generate the ITC, and $Q_F$ will be toggled from 0 to 1. This is the end of the count cycle. In mode 0/1, the $Q_F$ is available as $\overline{\text{TIMER OUT}}$, while in mode 2/3 $Q_F$ gated with the clock and TC is available as the $\overline{\text{TIMER OUT}}$ signal.

When the read operation is executed (after stopping the counter) on the timer, the two byte data obtained has the format illustrated in Fig. 15.16.

Thus if one is interested in knowing the remaining count, the data obtained cannot be directly interpreted to mean that it represents the remaining count (counter was stopped by stop command). The bits 1 to 13 signifies remaining count in an half cycle. Which half cycle was going on is indicated by bit 0($Q_F$). The remaining count is illustrated by Fig. 15.16.

To obtain remaining count, thus the following steps are to be carried out.

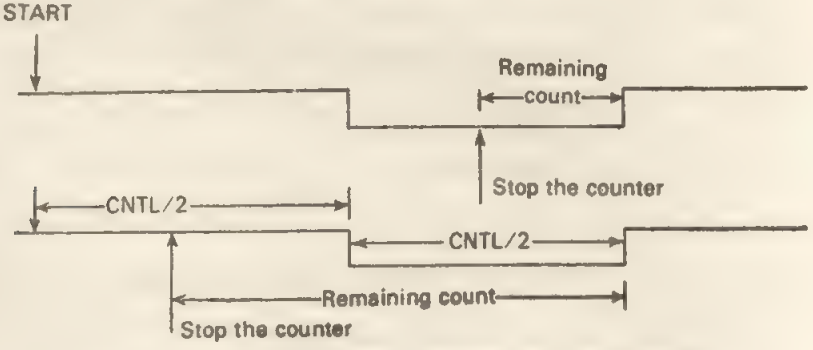



FIG. 15.15. REMAINING COUNT ILLUSTRATION.

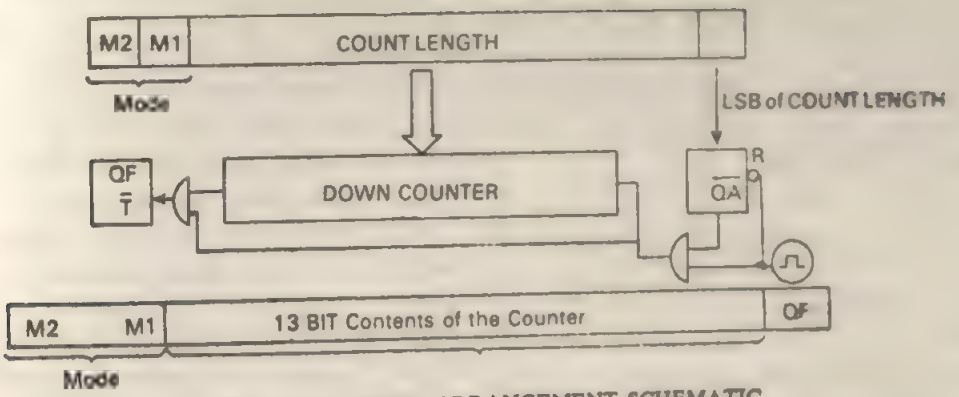



FIG. 15.16. TIMER ARRANGEMENT SCHEMATIC

1. Stop the counter.
2. Read the 16-bit timer data.
3. Remove mode bits (by ending 3FFFH).
4. Rotate the 16 bit through CY
5. If CY=0 the rotated quantity is the remain count.

else (CY=1), add CNTL/2 to the rotated quantity. The result shall give the remaining count.

**Example. Program** the timer to give out a square wave of period 500 count in pulses.

```
PROG   : LDA   FIVEH
         OUT   64H
         LDA   FIVEH+1
         OUT   65H
         ORI   40
         LDA   CW8156
         ANI   3F
         MOV   B, A
         LDA   TCOMD
         ORA   B
         OUT   60H ; start timer
         RLT
FIVEH  : DW    500
CW8156 : DB    xxxxxxB
         ; as per the port configurations
TCOMD  : DB    1100000B ; start timer
```

The programs could be written and tried out for the working of other modes of timer and I/O ports of 8155. It is left as an exercise to the readers.

### 15.4.5. Example on the 8155 Addressing

To clear the understanding of the access to the chip's facility, an example is presented in the present section, through which the addressing for 8155 is illustrated.

As mentioned earlier, the signals $\overline{CE}$ and IO/$\overline{M}$ along with internal 8 bits of address lines $A_7 - A_0$ (internal) determine the addressible entity.

**Memory Mapped I O**

Assume $A_{15}$ of address bus is connected to IO/$\overline{M}$ and $\overline{CS} = A_{14}\, A_{13}\, A_{12}.A_{11}.A_{10}\, A_9.A_8$ Thus $\overline{CS}$ is '0' when HOB of address bus has X110 0000 (X=don't care) pattern.

This arrangement will address memory section of 8155 when $A_{15}$=0 *i.e.*, 0110 0000 aaaa aaaa on the address bus. here, aaaaaaaa gives the location of RAM in the 8155, the RAM will thus have addresses

60aa (aa is the 8 bit address of location in RAM)

The $A_{15}$ line carrying a '1' will address I/O section of the chip. Thus the addresses will have the following bit patterns for accessing the I/O section

1110 0000 xxxxxaaa (X=don't care bit)

Thus the locations (addresses) E0X0, E0X1, E0X2, E0X3, E0X4, E0X5 will address the control/status, port A, port B, port C time LOB and timer HOB respectively during any instruction refering to these addresses. Here x is don't care Hex digit.

The chip is supposed to be used only in the memory mapped mode for I/0 section with the above arrangement. But it could be accessed by IN/OUT instructions partly as a side effect as follows.

Any address in the IN or OUT instruction with 60 or EO shall generate chip select active (due to same 8-bit byte appearing on the LOB and HOB of the address bus) and thus IN 60H will select chip for memory section and read data from the location 6060 *i.e.*, RAM location 60 of 8155. Similarly OUT 60H will write into the RAM location 60H of 8155. IN EO will select chip for I/O and get the data from the status register. OUT EO H similarly will put the data in the control register. Since chip will not be selected for other addresses, no data transfer can be done using IN or OUT with addresses other than 60H or EO.

## 15.5. THE 8355 ; ROM AND I/O

The 8355 provides 2K bytes of ROM and two 8-bit I/O ports whose individual I/O pins could be programmed as input pins or output pins. The I/O is basically in the unstrobed mode, *i.e.*, a byte of data could be written or read from these ports without control mechanism for data ready and other signals, *i.e.*, equivalent to mode 0 of 8255. The block diagram of the 8355 is shown in Fig. 15.17.

The chip has 2K bytes of ROM which is accessed by an 11 bit Address when both the chip selects are active and IO/$\overline{M}$=0. Like 8155,

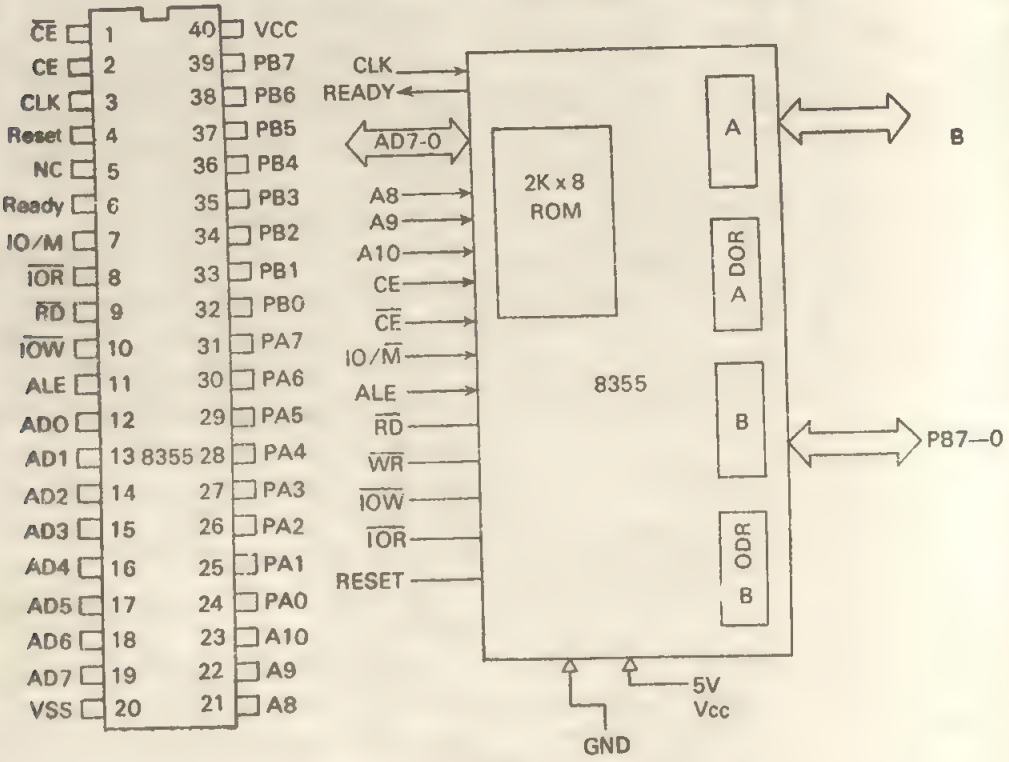



FIG. 15.17. 8355 ROM+I/O

the 8355 has internal latches for $\overline{CE_1}$ $\overline{CE_2}$, and IO/$\overline{M}$ and $AD_7$-$AD_0$. These signals are latched by ALE signal into the 8355. The memory section is accessed when both the chip selects are active and IO/$\overline{M}$=0. The 11-bit address required to specify a location is available to the memory section (3 bits) from the external address pins $A_8$, $A_9$ and $A_{10}$, while the 8-bits from the lower ordered address byte latched into the chip by the ALE signal). The memory section gives out data in response to the signal $\overline{RD}$.

The I/O section is accessed when IO/$\overline{M}$=1 and the chip selects are active.

The bits $A_1$ $A_0$ of the address are used to address the four registers as shown below.

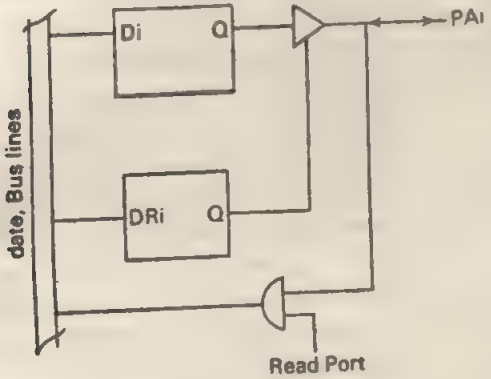



Fig. 15.18. Read Port.

The DDR bits control the directions of the respective pins. The schematic of the port A with DDR controlling the direction is shown in Fig. 15.18. The configuration of a port pin is done as output when DDR bit is 1. The various other actions are visible from the Fig. 15.18.

The following program segment sets the ports A and B of 8355 in the pinwise I/O mode. The first two pins of port A $PA_1$, $PA_0$ are to be configured in the input mode, Pin $PA_2$ in the output mode while the rest of them are in the input mode. Port B is to be configured in the output mode.

```
RESET : LDA  CWA
        OUT  62H
        LDA  CWB
        OUT  63H
        RET
CWA   : DB   00100000B
               (A port DDR setting)
CWB   : DB   11111111B
               (B port DDR setting)
```

The data from these ports now can be transferred if required. Since port A has individual pins configured, we shall keep the data word known to the program, which is present in the port, and modify the bits of this data byte to reflect new values on certain bits if required.

Also the port could be read to ascertain the values. The data read will reflect, the data on the I/O pins (whatever may be its direction).

## 15.6. THE 8755 EPROM AND I/O

The 8755 is same functionally as 8355 except that 8355 is ROM (already programmed, usually it holds 8085 system monitor) while 8755 is user programmable ROM. The 8755 has 1 additional pin (pin 5) which is unused in 8355 to be connected to Vdd for programming. The $\overline{CE_1}$ pin acts as a program

| $A_1$ | $A_0$ | Port Register | DDR bit ; | Direction |
|---|---|---|---|---|
| 0 | 0 | Port A | 1 ; | th bit of port in OUTPUT |
| 0 | 1 | Port B | 0 ; | th bit of port in INPUT |
| 1 | 0 | Port A Data Direction Register (DDR) | | |
| 1 | 1 | Port B Data Direction Register (DDR) | | |

pin. The chip is programmed by addressing the memory section $CE_2$ is 1, $IO/\overline{M}=0$ and the address supplied to it. A program pulse of 50 ms duration is applied to PROG/$\overline{CE_1}$ pin and at the same time Vdd is kept at 25V. The programming is done selectively byte wise. A program followed by read is advised. If any discrepancy is found in the programmed byte, it is programmed again.

## EXERCISES

1. Write a program to configure the 8155 for connecting ADC. Give your hardware schematic.
2. Design a keyboard encoder using 8155. Give complete hardware and program details.
3. Write programs to configure and use the timer in 8155 in various modes.
4. Give hardware and program for using 8255 in any application you may think.
5. The 8085 microprocessor does not support multiplication operation in its instruction set. It is desired to attach an 8×8 hardware multiplier (which multiplies two 8 bit unsigned numbers to produce a 16 bit unsigned result) to the system through I/O.

(*a*) Using appropriate peripheral chip/chips give the hardware design. Show only peripheral side of the lines and their usage.

(*b*) Give the peripheral initiallization program required to operate the above.

(*c*) Write a program using the above hardware to multiply two 16 bit unsigned numbers.

6. Write a program to transfer 100 butes of data from Port A (strobbed input mode) to Port B (unstrobbed output mode) of 8155 using interrupt data transfer. Show the hardware connections to obtain the interrupt and other control signals for the 8155 operation on the device side. Also write a Reset program to configure the 8155 for the above job.

# BIBLIOGRAPHY


**Chapter 1**

Bowden, B.V., "Faster than Thought", Putham and Co. Ltd., London 1953.

Donovan, J.J., "Systems Programming", McGraw—Hill, Kogakusha 1974.

Rosen, S., "Electronic Computers : A Historical Survey", ACM Computing Surveys, Vol 1, No. 1 March 1969.

Goldstine, H.H., "The Computer from Pascal to Von Neumann", Princeton Univ. Press, Princeton N.J. 1972.

Wilkes, M.V. and et al, "The Preparation of Programs for an Electronic Digital Computer", Addison-Wesley, Mass. 1951.

**Chapter 2**

Chu, Y., "Digital Computer Design Fundamentals", McGraw—Hill, N.Y. 1962.

Knuth, D.E., "The Art of Computer Programming", Vol. 2, Addison—Wesley, Mass 1969.

Kohavi, Z., "Switching and Finite Automata Theory", McGraw—Hill, N.Y. 1970.

**Chapter 3**

Karnaugh, M., "The Map Method for Synthesis of Combinational Logic Circuits", AIEE Trans. Part I, Vol. 72 pp. 593-599, 1953.

Kohavi, Z., "Switching and Finite Automata Theory", McGraw—Hill, N.Y. 1970.

McCluskey, E.J. Jr., "Minimisation of Boolean Functions", Bell Syst. Tech. Journ. Vol. 35, No. 6, pp 1417-1444, Nov. 1956.

Quine, W.V., "A Way to Simplify Truth Functions", American Maths Monthly, Vol. 62, No. 9, pp 627-631, Nov. 1955.

**Chapter 4**

Booth, T.L., "Digital Networks and Computer Systems", Wiley, N.Y. 1971.

Hellerman, H., "Digital Computer System Principles", McGraw—Hill, N.Y. 1971.

Lee, S.C. "Digital Circuits and Logic Design", Prentice—Hall of India, 1978.

Millman, J. Halkias, "Integrated Electronics", McGraw—Hill, Kogakusha 1972.

Morns, R.L. and Miller, J.R., "Designing with TTL Integrated Circuits", McGraw—Hill, N.Y. 1971.

Taub H. and Schilling, D., "Digital Integrated Electronics", McGraw—Hill 1977.

Wickes, W.E., "Logic Design with Integrated Circuits", Wiley, N.Y. 1968.

**Chapters 5, 6, 7, 8**

Chu, Y., "Digital Computer Design Fundamentals", McGraw—Hill, N.Y. 1962.

Flores, I., "The Logic of Computer Arithmetic", Prentice Hall Inc., Englewood Cliffs, N.J. 1963.

Gschwind, H.W., "Design of Digital Computers", Spriger-Verlog, N.Y. 1967.

Richards, R.K., "Arithmetic Operations in Digital Computers", D. Van Nostrand Comp. Inc., Princeton, N.J. 1950.

Stein, M.L. "Introduction to Machine Arithmetic", Addison-Wesley 1972.

**Chapter 10**

Bremer, J.W., "A survey of Main-frame Semiconductor Memories", Computer Design, pp 63-73, May 1970.

Bryant, R.W. and et al, "A High Performance LSI Memory System", Computer Design, pp 71-77 July 1970.

Hodges, D.A., Ed., "Semiconductor Memories", IEEE Press.

**Chapter 11**

Bell, C.G. and Newell, A., "Computer Structures : Readings and Examples", McGraw—Hill, N.Y. 1971.

Chu, Y., "Introduction to Computer Organisation", Prentice—Hall, Englehood Cliffs, N.J. 1970.

Flores, I., "Computer Organisation", Prentice—Hall, Englewood Cliffs, N.J. 1966.

Hayes, J.P., "Computer Architecture and Organisation", McGraw—Hill, Kogakusha, 1978.

Hellerman, H., "Digital Computer System Principles", McGraw-Hill, N.Y. 1967.

Tanenbum, A.S., "Structured Computer Organisation", Prentice—Hall of India, New Delhi 1979.

**Chapter 12**

Abd—Alla, A.M. and Meltzer, A.C., "Principles of Digital Computer Design", Prentice—Hall, Englewood Cliffs, N.J. 1976.

Chu, Y., "Computer Organisation and Microprogramming", Prentice—Hall, Englewood Cliffs, N.J. 1970.

Hayes, J.P., "Computer Architecture and Organisation", McGraw—Hill, Kogakusha, 1978.

Hussan, S.S., "Microprogramming Principles and Practice", Prentice—Hall, Englewood Cliffs, N.J., 1970.

Wilkes, M.V., "The Growth of Interest in Microprogramming, "ACM Computing Surveys, pp 139-145, 1969.

**Chapter 13**

Bartee, T.C., "Digital Computer Fundamentals", McGraw—Hill Kogakusha, 1977.

Flores, I., "Peripheral Devices", Prentice—Hall, Englewood Cliffs, N J.

Garner, H.L., "Generalised Parity Checking", IRE Trans. on Electronic Computer, EC-7, No. 3, pp 207-213, Sept. 1958.

Hamming, R.W., "Error Detecting and Error Correcting Codes", Bell Syst. Tech. Journ. Vol. 29, pp 147-160 April 1950.

Hayes, J.P., "Computer Architecture and Organisation", McGraw—Hill Kogakusha, 1978.

Lin S., "An Introduction to Error Correcting Codes", Prentice—Hall, Englewood Cliffs, N.J. 1967.

Peterson, W.W., "Error Correcting Codes", MIT press, Mass. 1961.

Sellers, F.F., Hsiao, M.Y. and Bearsnson, L.W., "Error Detecting Logic for Digital Computers", McGraw—Hill, N.Y. 1968.

**Chapters 14, 15**

"MCS—80 User's Manual (With Introduction to MCS—85)", Oct. 1977 Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051.

"MCS—85 User's Manual, Intel Corporation", Oct. 1978.

"The 8086 Family User's Manual, Intel Corporation", Oct. 1979.

"Intel Component Data Catalogue", Intel Corporation 3065 Bowers Avenue, Santa Clara, CA 95051, 1985.

"Microprocessor and Peripheral Handbook" Intel Corporation 1985.

# Index